# PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS

Conor Cunningham, César A. Galindo-Legaria, Goetz Graefe
*Microsoft Corporation*
*One Microsoft Way*
*Redmond, WA 98052 USA*
*{conorc,cesarg,goetzg}@microsoft.com*

## Abstract

*PIVOT and UNPIVOT, two operators on tabular data that exchange rows and columns, enable data transformations useful in data modeling, data analysis, and data presentation. They can quite easily be implemented inside a query processor, much like select, project, and join. Such a design provides opportunities for better performance, both during query optimization and query execution. We discuss query optimization and execution implications of this integrated design and evaluate the performance of this approach using a prototype implementation in Microsoft SQL Server.*

## 1. Introduction

Pivot and Unpivot are complementary data manipulation operators that modify the role of rows and columns in a relational table. Pivot transforms a series of rows into a series of fewer rows with additional columns. Data in one source column is used to determine the new column for a row, and another source column is used as the data for that new column. Unpivot provides the inverse operation, removing a number of columns and creating additional rows that capture the column names and values from the wide form. The wide form can be considered as a matrix of column values, while the narrow form is a natural encoding of a sparse matrix. Figure 1 demonstrates how Pivot and Unpivot can transform data between narrow and wide tables. For certain classes of data, these operators provide powerful capabilities to RDBMS users to structure, manipulate, and report data in useful ways.

Implementations of pivoting functionality already exist for the purpose of data presentation, but these operations are usually performed either outside the RDBMS or as a simple post-processing operation outside of query processing. Microsoft Excel, for example, supports pivoting. Users can perform a traditional SQL query against a data source, import the result into Microsoft Excel, and then perform pivoting operations on the results returned from that data source. Microsoft Access (which uses the Microsoft Jet Database Engine) also provides pivoting functionality. This pivot implementation is a post-processing operation through cursors. While existing implementations are certainly useful, they fail to consider Pivot or Unpivot as first-class RDBMS operations, which is the topic of this paper.

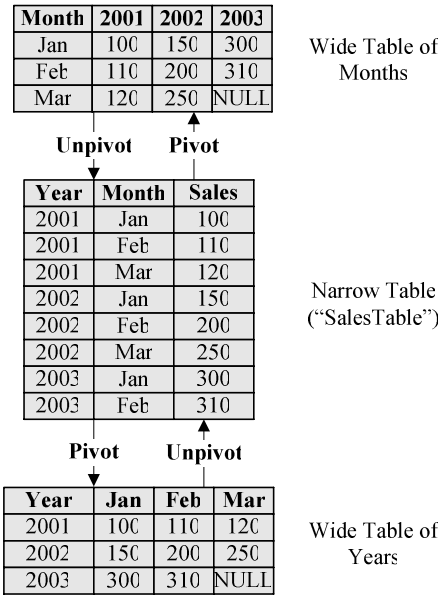



**Figure 1 Pivot and Unpivot**



**Proceedings of the 30th VLDB Conference,**
**Toronto, Canada, 2004**

Inclusion of Pivot and Unpivot inside the RDBMS enables interesting and useful possibilities for data modeling. Existing modeling techniques must decide both the relationships between tables and the attributes within those tables to persist. The requirement that columns be strongly defined contrasts with the nature of rows, which can be added and removed easily. Pivot and Unpivot, which exchange the role of rows and columns, allow the *a priori* requirement for pre-defined columns to be relaxed. These operators provide a technique to allow rows to become columns dynamically at the time of query compilation and execution. When the set of columns cannot be determined in advance, one common table design scenario employs "property tables", where a table containing (id, propertyname, propertyvalue) is used to store a series of values in rows that would be desirable to represent columns. Users typically use this design to avoid RDBMS implementation restrictions (such as an upper limit for the number of columns in a table or storage overhead associated with many empty columns in a row) or to avoid changing the schema when a new property needs to be added. This design choice has implications on how tables in this form can be used and how well they perform in queries. Property table queries are more difficult to write and maintain, and the complexity of the operation may result in less optimal query execution plans. In general, applications written to handle data stored in property tables can not easily process data in the wide (pivoted) format. Pivot and Unpivot enable property tables to look like regular tables (and vice versa) to a data modeling tool. These operations provide the framework to enable useful extensions to data modeling.

Item Table

| ItemKey | Item Name |
|---|---|
| 1 | 2001 |
| 2 | 2002 |
| 3 | 2003 |

Property Table

| ItemKey | PropName | PropValue |
|---|---|---|
| 1 | Jan | 100 |
| 1 | Feb | 110 |
| 1 | Mar | 120 |
| 2 | Jan | 150 |
| 2 | Feb | 200 |
| 2 | Mar | 250 |

**Figure 2 Property Table**

Including Pivot and Unpivot explicitly in the query language provides excellent opportunities for query optimization. Properly defined, these operations can be used in arbitrary combinations with existing operations such as filters, joins, and grouping. For example, since Unpivot transposes columns into rows, it is possible to convert a filter (an operation that restricts rows) over unpivot into a projection (an operation that restricts columns) beneath it. Algebraic equivalences between Pivot/Unpivot and existing operators enable consideration of many execution strategies through reordering, with the standard opportunity to improve query performance. Furthermore, new optimization techniques can also be introduced that take advantage of unique properties of these new operators. Consideration of these issues provides powerful techniques for improving existing user scenarios currently performed outside the confines of a query optimizer.

We argue that pivoting operations can be performed more quickly and powerfully inside a RDBMS. By implementing these operations as relational algebra operators within a cost-based optimization framework, superior execution strategies can be considered. This design choice also allows other relational operations to be performed on the results of pivot and unpivot. Considerations of the interactions between pivot/unpivot and other operators yield more efficient orderings of operations over post-processing. The inclusion of these operations within the declarative framework of a SQL statement also allows consideration of additional access paths, such as indexes or materialized views, to more efficiently compute results. Consideration of Pivot and Unpivot within a cost-based optimizer framework provides opportunities for superior performance over existing approaches.

The rest of this paper is organized as follows: Section 2 defines Pivot and Unpivot syntax and semantics as well as useful variations. Algebraic optimizations are covered in Section 3, followed by execution considerations in Section 4. An implementation and evaluation of these operators using Microsoft SQL Server is performed in Section 5. Possible extensions are discussed in Section 6, followed by related work and conclusions.

## 2. Introducing Pivot and Unpivot

### 2.1. PIVOT and UNPIVOT in SQL

It is possible to implement pivoting in standard SQL, though the syntax is cumbersome and its performance is generally poor. One method to express pivoting uses scalar subqueries in the projection list. Each pivoted column is created through a separate (but nearly identical) subquery as seen in Figure 3. For database implementations that do not support PIVOT, users could employ this technique to perform pivoting operations. (Note that SalesTable is defined graphically in Figure 1).

```
SELECT
Year,
(SELECT Sales FROM SalesTable AS T2 WHERE Month = 'Jan' AND T2.Year = T1.Year) AS 'Jan'
(SELECT Sales FROM SalesTable AS T2 WHERE Month = 'Feb' AND T2.Year = T1.Year) AS 'Feb'
(SELECT Sales FROM SalesTable AS T2 WHERE Month = 'Mar' AND T2.Year = T1.Year) AS 'Mar'
FROM SalesTable AS T1
GROUP BY Year
```

**Figure 3 Possible PIVOT Syntax**

Unfortunately, this approach has limitations that restrict the power of pivoting. Each column has repetitive syntax, which is cumbersome as the number of pivoted columns increases. These syntaxes are also potentially harder to optimize. For this syntax, the query optimizer is presented with a number of subqueries, making it more difficult to determine that this whole operation represents a "Pivot" on a single table. In practice, this is not an easy operation, making pivot-specific optimizations very difficult. The common problem is that the *intent* of the query is difficult to infer from the syntax or common relational algebra representation.

Therefore, we propose the following syntax for PIVOT in Figure 4 as an additional option under the <table expression> rule of the ANSI SQL grammar. This syntax is easier to read and better captures the intent of the desired operation. Repetition is eliminated, making queries easier to ready, write, and maintain. Section 3 shows that this approach also enables additional query optimization techniques.

```
SELECT * FROM
(SalesTable PIVOT (Sales for Month IN ('Jan','Feb','Mar'))
```

**Figure 4 PIVOT Syntax**

PIVOT operates on a table, like other operations, converting from narrow form to wide form. The column 'Sales' in SalesTable provides values for the pivoted columns, while the values of the Month column define the mapping describing in which column the value from Sales belongs. The IN list describes the values of interest from the Month column as well as the names of the new columns to create in PIVOT. The remaining columns from SalesTable, though not listed, implicitly divide the rows of SalesTable into groups. Each group of rows becomes a single output row as a result of PIVOT.

For Unpivot, we propose similar syntax to undo the pivoting operation. The UNPIVOT syntax in Figure 5 contains the same major elements. The set of columns to be removed are listed in the IN list, and the two new columns to create are listed (Sales and Month in this example). While PIVOT collapses similar rows into a single, wider row, UNPIVOT does the opposite. The operation multiplies the number of rows by the number of elements in the IN list while reducing the number of columns.

```
SELECT * FROM
(SalesReport UNPIVOT (Sales for Month IN ('Jan', 'Feb', 'Mar'))
```

**Figure 5 UNPIVOT Syntax**

## 2.2. PIVOT and UNPIVOT Semantics

While the conceptual model for PIVOT and UNPIVOT is straightforward, several important details must be further defined to operate well with existing SQL constructs. One problem that must be addressed is how to handle data collisions (two values mapping to the same location). Missing values is the opposite condition, and behavior must also be defined for this case. Finally, the use of PIVOT and UNPIVOT on dynamic (open) schemas must be addressed. Any Pivot and Unpivot definitions must handle these semantic issues.

Data collisions are possible and can be handled in a number of ways. It is possible to error on collisions, though this requires special run-time logic in a query plan to enforce the behavior. It may be useful to pivot data that has duplicate values, and adding a collapsing function (such as an aggregate) enables PIVOT to work in this scenario. In Figure 6, the PIVOT syntax is extended to handle collisions through the SUM() aggregate. Avoidance of collisions is also possible through a special constraint that precludes duplicates from being introduced at all. For example, if the grouping columns and the pivot column (Sales, in this example) together form a unique key, then PIVOT is guaranteed not to have any collisions. Still another strategy could involve nested result sets, where all values are preserved in nested tables in the output of PIVOT. All of these strategies are effective techniques in our implementation to resolve any ambiguity of the PIVOT operation.

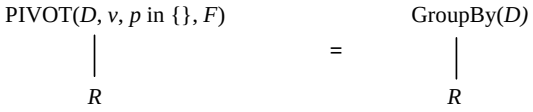
```
SELECT * FROM
(SalesTable PIVOT (SUM(Sales) for Month IN ('Jan', 'Feb', 'Mar'))
```

**Figure 6 PIVOT Syntax with Aggregation**

Missing values as a result of both PIVOT and UNPIVOT is the complimentary condition to data collisions. For PIVOT, it is possible just to use NULL to represent this condition. However, NULLs are also a valid output, leading to the problem of disambiguating which NULLs were introduced by the PIVOT operation. This problem is also seen in operations such as CUBE [4], and can be handled by a special disambiguating function that outputs whether the row was introduced in the operation. Another technique to handle the absence of values exists if a collapsing function (such as an

Let *R* be the input relation.
Let *D* be the set of columns from *R* that define groups.
Let *p* be a column in *R* not in *D*. Its value is the name of the new column to create when pivoting.
Let *v* be a column in *R* not in *D* where *p* not equal to *v*.
Let *F* be a collapsing aggregate function.
R' is a copy of R, with columns D', p', v' corresponding to D, p, and v respectively.
Let X by the set of columns $w_{1..k-1}$ representing the set of pivoted columns.
PIVOT has result columns *D* plus columns $w_{1..k}$ representing the pivoted values.

Base Case (no pivoted rows):




Inductive Case (one or more pivoted value(s) x):

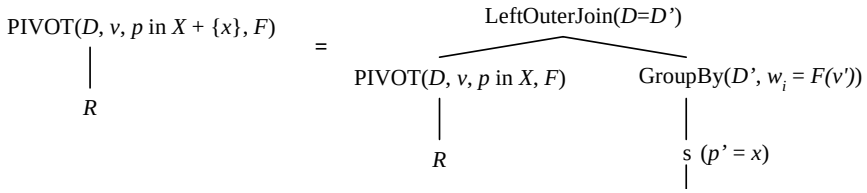



**Figure 8 PIVOT Definition**

aggregate) is used. In this case, it is possible to treat this as an empty set, returning whatever the empty aggregate result would be. For COUNT(column), this would be zero. UNPIVOT is relatively simple - it transposes the values in columns into their own rows, so no new NULLs are introduced. UNPIVOT can be defined to preserve or eliminate NULL values when generating rows.

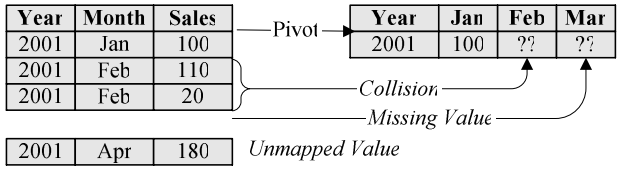



**Figure 7 Collisions and Empty Values**

PIVOT and UNPIVOT may or may not preserve data, based on how they are defined and used in queries. To fully preserve data, these operations must be defined to avoid collisions and to not introduce empty values (NULLs). Furthermore, "missing" values in the PIVOT IN list are implicitly removed, acting like a projection. PIVOT and UNPIVOT are not inverses if their IN lists do not cover the complete set of data values in the pivot operation, so care is required to preserve data when using PIVOT and UNPIVOT. As a whole, avoiding these restrictions would reduce the flexibility of these operators, so extending PIVOT and UNPIVOT to work on non-invertible data enables broader application to operations beyond inversing data.

While there is no formal database mechanism to enforce that PIVOT and UNPIVOT be used in a data-preserving, invertible fashion, it is possible to get most of this capability through CHECK constraints. By restricting the pivot column to a list of valid values, the query optimizer can infer whether the PIVOT operation is data-preserving if its IN-list matches the constraint. This could be further extended by either limiting PIVOT and UNPIVOT to cases when such constraints exist or through the creation of a stronger class of constraint in the RDBMS.

In the syntax proposed in this paper, the pivot columns are explicitly defined in the query. If PIVOT were to generate output columns at runtime (i.e. late binding), this would introduce problems about how references would be resolved for query operators in a tree. Typically, SQL queries must define the list of columns at compile time to allow the user know the set output columns before running the query. If PIVOT exists below other query operators (as this client syntax allows), it also would cause problems for existing operators that expect a fixed set of columns (i.e. distinct). The actual limitations imposed by this restriction are small, as most database systems support transactions with multiple commands that could be used to build the current list of columns and then pivot on them in separate queries, maintaining the existing strongly bound semantic.

## 3. Algebraic Optimizations

Queries containing PIVOT and/or UNPIVOT have the opportunity to perform better if interactions with existing operators (filters, projections, join, etc.) are considered. Algebraic rewrites of PIVOT/UNPIVOT and other operators enable cost-based optimizers to consider alternative execution strategies to find more optimal plans. This section covers some basic rewrites related to filters and projections, more complicated transformations converting PIVOT or UNPIVOT into existing operators, using pivoting efficiently in property tables, and the introduction of PIVOT and UNPIVOT into queries that contain neither.

As a note about the terminology used in this section, this paper assumes that duplicates work as in SQL. As a result, Project and Union operations preserve duplicates and are cardinality-preserving. Group By operations are used to distinct values and can also be used to compute aggregate functions.

We formally define PIVOT in Figure 8 by defining PIVOT without any pivoted columns as DISTINCT, and then inductively add pivoted columns to the base definition through a left outer join to calculate the pivoted values.

UNPIVOT is defined using the same variables used in the definition of PIVOT. It is defined as an Apply over the Union of a series of row constructors (one for each column to be unpivoted).

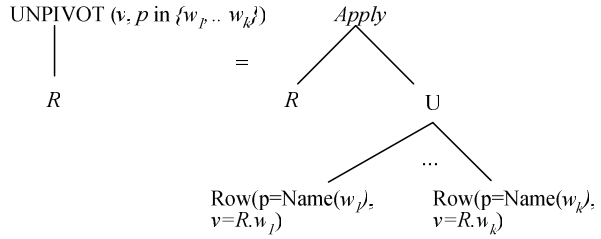


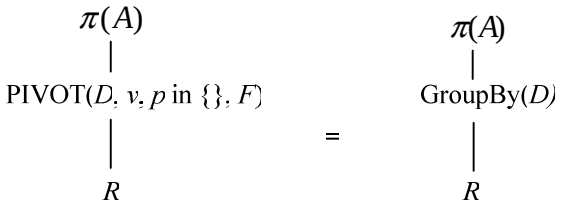

**Figure 9 UNPIVOT Definition**

### 3.1. Projections and Filters

Projections and Filters are both restricting operations on different dimensions of relations (one restricts columns, while the other restricts rows). Interestingly, since PIVOT and UNPIVOT exchange rows and columns, this provides an opportunity to transform a projection to and from selections. This section describes different PIVOT/UNPIVOT algebraic rewrites invoking projection and selection.

Projections can be used to simplify PIVOT and avoid unnecessary computation. If a query uses a Project to restrict column(s) introduced by PIVOT, the query can be rewritten to not PIVOT those columns at all. This simplification is possible since each pivoted column is independent from all others.

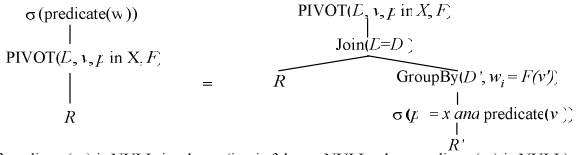
It might be assumed this class of projection also implies that a filter could be introduced below pivot to restrict the pivot column (Month, in this example) to be limited to 'Jan' or 'Feb'. Unfortunately, this is not true. Semantically, PIVOT produces a row for each group even when input rows exist that do not match the pivot column list. Pushing such a filter would eliminate groups with no pivoted values, and it therefore does not work in the general case. There are situations when filters can be used, and these are described later in this section.




If A is a subset of D (i.e. no pivoted columns)

**Figure 10 PIVOT Projection Pushdown Identity**

Filters over columns created in PIVOT can be pushed in some cases. If there are guaranteed to be no duplicates (if the grouping columns and the pivot column together comprise a key) and the collapsing function is the identity, it is possible to use the technique described in Figure 11. One scan of the input finds any value matching the filter criteria, and then a join is performed with another instance of the input to gather the remaining columns to complete the PIVOT operation. This technique is most beneficial if indexes are defined that allow efficient searching of these tables.




If predicate(w) is NULL-intolerant(i.e. is false or NULL when predicate(w) is NULL), D+{p} is a key of R, and F(v) = v

**Figure 11 PIVOT Filter Pushdown**

Projections over UNPIVOT are straightforward. Projections limiting grouping columns can be safely applied below the UNPIVOT. A projection removing the value column implies that none of the pivoted columns are actually needed to perform the UNPIVOT. It is still necessary to perform some transformation in this case to generate the correct number of duplicates of each input

row to UNPIVOT. However, this transformation could actually performed by UNION ALL instead of UNPIVOT.

Filters interact with UNPIVOT in a similar fashion to how Projects interact with PIVOT. A filter on the columns introduced by UNPIVOT enables a whole column to be removed from the input to UNPIVOT. UNPIVOT does not have the same problem PIVOT faces in preserving groups since it operates on columns. Columns listed for transformation are all transformed, and columns not listed become grouping columns. Thus, data is preserved in all cases and a projection can be safely introduced below the UNPIVOT.

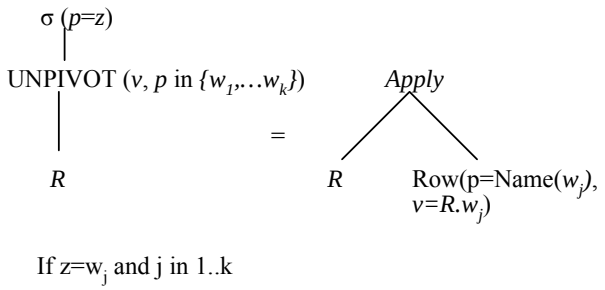



**Figure 12 Filters Interact with UNPIVOT**

The PIVOT syntax described in Figure 4 contains an IN list describing the set of values used to create new pivoted columns. This limits the set of interesting rows to rows that have a column value in this IN list, as other values would be ignored by PIVOT. While it seems possible to use this IN list to introduce an implied filter under PIVOT, it does not preserve all groups correctly. The following section describes a scenario when an implied filter can be used.

## 3.2 PIVOT and Property Tables

PIVOT is useful in data modeling because they can hide the physical storage design and provide a consistent "wide" format to the rest of a database. In a typical scenario, two tables are used, storing a list of items in one table and all its properties with their values in the other. In terms of PIVOT, the grouping columns are delivered from one table, while the pivot and value columns are delivered from the other. Property tables contain property name and value columns that represent the sparse matrix of (column, value) pairs of the virtualized table. PIVOT can transform this physical representation into a virtual table containing all the columns (with NULLs in any missing locations). As these tables are typically joined together using a left outer join on a set of key columns (matching the grouping columns in PIVOT), it is possible to perform transformations on this structure to improve plan selection.

While the additional complexity of this design does have some overhead, the overall impact can be minimized through proper plan and index selection. In most cases, creating indexes over the grouping columns on the item table as well as the property and value columns of the property table enable index lookup plans. One observation about query transformations in this design is that pushing projections and filters will not always produce superior plans. Some transformations require additional scans of the input, so they will only be beneficial if the proper indexes exist and predicates are sufficiently selective.

When used against property tables, a projection that removes all pivoted columns can be simplified as in Figure 13. PIVOT becomes a Distinct over the left outer join between the item and property table. However, since no columns are used from the property table, that join can be removed. Furthermore, since the property table design typically has a key over the grouping columns in the item table, the complete pattern can be satisfied with a scan of the item table.

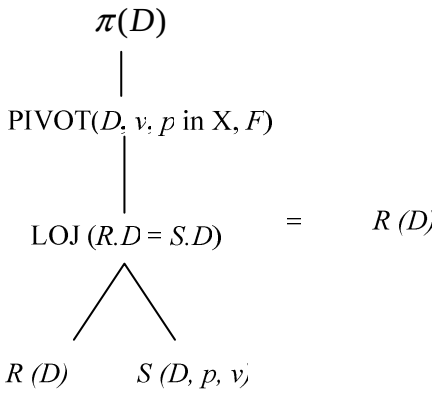
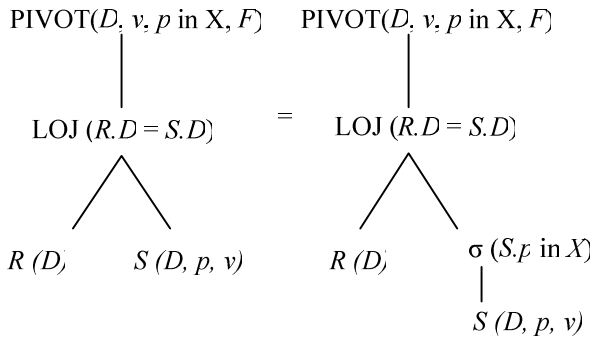



**Figure 13 Property Table Projection Reduction**

A filter can be implied from the IN-list in PIVOT when used against the property and item tables. Since groups are preserved by the outer join, a filter can be introduced below the join to restrict the property table to only have property names in the set of columns being pivoted. In Figure 13, the grouping columns are delivered by R, while the property table is S. A left outer join between these tables allows all the properties from the groups listed in R to be surfaced above the join. However, it is known that PIVOT will only consume property values (from S) if they are in the IN-list of the

PIVOT. Therefore, properties can be pre-filtered on S before the join.

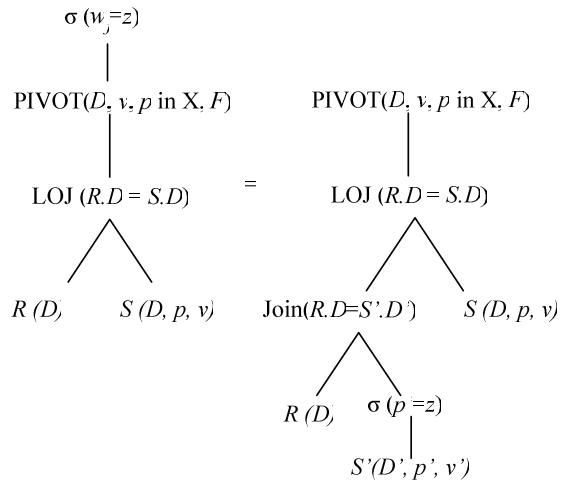



If D+{p} is a key of S and D is a key of R

**Figure 14 Implying a Filter from PIVOT**

Projections restricting the set of pivoted output columns from PIVOT also can introduce a filter on the property table. Since such a projection is equivalent to not pivoting the extra columns at all, the set of pivoted columns can be reduced, and the introduction of a filter follows from Figure 14.

Filters over PIVOT can be pushed, but only with the introduction of an additional scan of the property table. In Figure 15, a filter on a pivoted column can be rewritten to restrict the item table (R) to only consider items that have qualifying properties. Since this rewrite introduces an additional scan of the property table, this may be appropriate only when certain indexes are defined on the item and property tables.




If D+{p} is a key of S, S' is a copy of S, and D is a key of R

**Figure 15 Filter Pushdown on Property Tables**

### 3.3 PIVOT as GROUP BY

It is possible to rewrite queries using PIVOT to instead use GROUP BY. Each value in the IN-list uses a copy of the aggregate function listed in the PIVOT definition. Beneath each aggregate, conditional logic is used to pick only the input rows that map to the correct output column. Non-matching rows are changed to NULL instead. Columns not listed in PIVOT become the set of grouping columns for the GROUP BY. The syntax from Figure 6 maps as follows:

```
MIN(CASE Month WHEN 'Jan' THEN Sales ELSE NULL END) AS 'Jan',
MIN(CASE Month WHEN 'Feb' THEN Sales ELSE NULL END) AS 'Feb',
MIN(CASE Month WHEN 'Mar' THEN Sales ELSE NULL END) AS 'Mar'
```

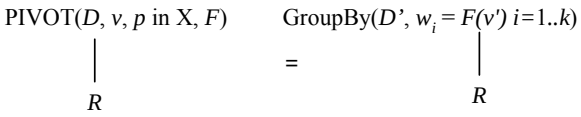



**Figure 16 PIVOT Identity**

While the transformation to GROUP BY is straightforward, there are very good reasons to perform this step during the optimization of the query instead of as part of the declarative SQL definition. Queries defined using a series of aggregates in a GROUP BY are typically much harder for optimizers to examine and understand. Using PIVOT, logic is not distributed over a number of aggregate functions and operators with additional non-trivial scalar logic in each. Therefore, it is easier for rule-based optimizers to target with special-purpose transformation logic. Additionally, the syntax suggested in this paper is far simpler than current workarounds using standard SQL.

Mapping PIVOT to GROUP BY requires an assumption that the collapsing (aggregate) function be invariant to additional NULLs. The scalar logic beneath each aggregate substitutes NULL for each row that does not match that pivoted column. Formally speaking, a collapsing function *F* needs to support the condition that for any set of input values S, F(S) = F(S U {NULL}). Aggregates such as SUM() and MAX() have this property. However, COUNT(*) does not have this property, as it counts each row in its output.

Since PIVOT is a specialization of GROUP BY, RDBMS implementations can leverage this information to

easily add PIVOT support without writing new logic throughout every portion of a query processor. Transforming PIVOT into GROUP BY early in query compilation (for example, at or near the start of query optimization or heuristic rewrite) requires relatively few changes on the part of the database implementer. With such an approach, no new execution operators are required, and little new optimization or costing logic is needed. This provides an effective technique to extend existing RDBMS products with little effort.

In addition to implementation simplicity, viewing PIVOT as GROUP BY also yields many interesting optimizations that already apply to GROUP BY. Reusing existing GROUP BY optimization logic can yield an efficient PIVOT implementation without significant changes to existing code. These benefits include:

- Removal of duplicate or grouping columns by other grouping columns, which reduces overall row width
- Filters and semi-joins restricting complete groups can be performed below the GROUP BY.
- Local/Global techniques [7] for pushing grouping optimizations below joins and other operations
- Query logic to perform groupings using parallel threads of execution.

### 3.4 UNPIVOT as Apply

UNPIVOT can leverage existing implementation code as well. As an operation that takes each row and returns a number of additional rows as output, this is very similar to a correlated join (which we call an *Apply* [1]). If a join is made with a special constant table containing one row for each column to unpivot, then one row for each pivot column will be created in the output. This technique allows database implementations to get good performance characteristics without implementing a full operator in the query processor. While this approach introduces some additional overhead by using multiple iterators, this has not been significant in our implementation. It is not difficult to write a special purpose iterator if needed.

This transformation yields similar performance benefits to implementing PIVOT as GROUP BY. Apply can be reordered easily with other join operators, and it has well-defined interactions with filters, projections, and other query operations. It is also possible to perform these operations in parallel by segmenting the input rows into different groups. None of these problems need to be explicitly considered for UNPIVOT, as they are already solved for the regular join case. This greatly reduces the required implementation effort while still providing excellent performance for the operation.

Figure 9 describes a possible implementation of UNPIVOT using Apply, UNION, and a number of special "Constant Tables". Each UNION branch generates one unpivoted row with two columns. One column contains the name of the original column, and another contains the value associated with that column. The grouping columns are added to this row in the Apply. The Apply needs to be a Left Outer Apply, preserving rows from the input table when the UNPIVOT specifies no matching column names from the relation.

### 3.5 Join Cardinality Reduction

If PIVOT and UNPIVOT are inverses, a query optimizer can introduce PIVOT and UNPIVOT into a query tree as a technique to reduce cardinality in portions of a query tree around expensive operations, such as joins. If PIVOT can be used to reduce the cardinality of the input in a lossless fashion, joins (or other expensive operations) would be executed far fewer times, followed by an UNPIVOT to expand rows back to their original state. Cost-based optimizers can then pick the cheaper technique for query evaluation. Figure 17 shows an example of this.

The PIVOT and UNPIVOT operations must preserve the rows from the pivoted table as if they were processed by the join to be used in this transformation. Furthermore, if PIVOT uses a collapsing (aggregate) function, then UNPIVOT must be able to invert it in all cases. This could be achieved through nested scalars or other complex data types, invertible aggregate functions, or just avoiding data collisions through constraints on the input relation.

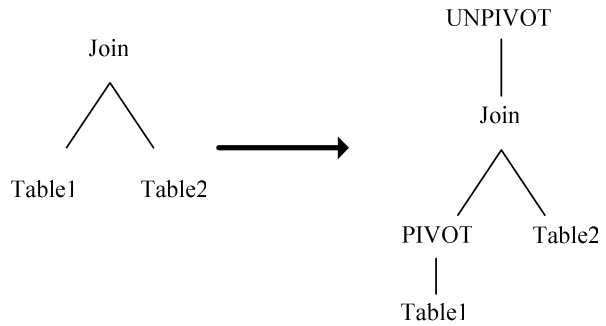



**Figure 17 PIVOT-based Join Cardinality Reduction**

PIVOT also provides the opportunity to represent a series of scalar subqueries (as seen in Figure 3) in a more semantically useful internal representation. A naive implementation would create a series of subqueries over the same table to compute each column. By converting this series of operations into PIVOT, the poor user representation can be handled with a far fewer number of

tables. This allows the database implementer to handle existing work-around queries generated by users before PIVOT existed.

## 4. Execution Strategies

Defining PIVOT in terms of GROUP BY and Apply provide an excellent opportunity to re-use existing execution operators in new ways. In Section 3.3, we demonstrated that PIVOT can be implemented as GROUP BY. Hash and stream aggregation are available for PIVOT, and have similar execution properties. Parallel query execution can also be supported using these execution strategies as long as the members of each group are processed in the same thread. PIVOT does use a relatively large number of identical aggregates with almost identical scalar logic. One novel execution strategy could group the computation of these aggregates together, either by treating the set of aggregates as a vector computation or by rewriting each individual aggregate computation into a dispatch table (as each column will be looking for a single and likely unique scalar for each input row).

PIVOT can also be implemented through a special-purpose iterator transposing rows into columns. Consuming a sorted (grouping columns and the pivot column) stream, the next row in the current group becomes the source of the value for the next column. If a pivoted column does not have a corresponding row in the input, it returns the empty value for all output columns until the correct location for the current input row is located. Similar to the grouping operators, this technique can be performed simultaneously over values from different groups.

As described in this paper, UNPIVOT can be implemented as a correlated nested loops join (Apply). Each invocation of the Apply can be performed in parallel, leveraging existing parallel techniques available to joins. UNPIVOT can also be implemented using a special purpose execution iterator that consumes one row and returns a number of rows in unpivoted form. Parallelism is slightly easier for UNPIVOT since each input row can be processed independently (instead of groups of rows).

## 5. Experimentation

We implemented PIVOT and UNPIVOT in Microsoft SQL Server, adding support in the parser and in the query processor for these new operators. The architecture of our query optimizer is based on the Cascades framework [3], which enables defining new relational operators and optimization rules for them. These optimization rules follow from the properties described earlier for PIVOT and UNPIVOT.

In this section we go over a number of scenarios and show the performance obtained in our system. We use the well-known TPCH database, at 1 GB scale, as a basis for our experiments. The experiments were conducted on a dual-processor machine running at 2 Gigahertz, with 1 GB of main memory. We flush data caches before executing queries, so the numbers shown are on a cold cache. Parallel execution is disabled in the results we present, since it does not qualitatively affect our results.

### 5.1. PIVOT vs. SQL sub-query form

We first compare the performance of our PIVOT operator with that of the equivalent formulation with sub-queries described in Section 2.1. The following query summarizes sales data in the ORDERS table, returning one row per year, and columns for each of the twelve months.

```
SELECT * FROM
 (SELECT
    YEAR(O_ORDERDATE),
    MONTH(O_ORDERDATE),
    O_TOTALPRICE
    FROM ORDERS) ORD(YEAR, MONTH, PRICE)
  PIVOT (SUM(PRICE)
  FOR MONTH IN (1,2,3,4,5,6,7,8,9,10,11,12)) T
```

Figure 18 shows the execution time of the PIVOT query and the equivalent sub-query formulation. For each of the two forms, we change the number of months to PIVOT; only three months of the year, then six months, and finally all twelve months. The performance difference is due to the duplication of work in the sub-query formulation, as each pivoted column is computed separately, because our common sub-expression code does not currently handle this case.

More indices can be used to speed up the computation of the sub-query form, even if the common sub-expression is not detected. Fast lookup of the value from the dimensions columns (e.g. and index on year, month, price in the case above) would make performance comparable to the PIVOT form. However, it remains verbose and repetitive to the application writer.

### 5.2. Property table access

Earlier, we mentioned the use of PIVOT to support property tables. This allows presenting a view of wide rows to application writers, even if a sparse representation

is used to store data internally. For this experiment, we added a property table to store information about the TPCH CUSTOMER table. The property table has the following schema:

```
CUSTPROPERTY(CP_CUSTKEY, CP_NAME, CP_VALUE)
```

Columns (CP_CUSTKEY, CP_NAME) make up a key of the property table. A customer property is registered in the database by inserting a new row to CUSTPROPERTY. We also create an index on CP_NAME, CP_VALUE, CP_CUSTKEY, to lookup property values efficiently.

We now create a view that exposes a "wide" customer row, having a column for property in a set of interest. The view uses an outer join to find the registered properties for the set of customers, because we want to retain customers even if no property is defined for them. Say we are only interested in five properties, 'A' through 'E':

```
CREATE VIEW EXTCUSTOMER AS
SELECT *
FROM (
   SELECT *
   FROM CUSTOMER LEFT JOIN CUSTPROPERTY
     ON C_CUSTKEY = CP_CUSTKEY
) CUSTNARROW
    PIVOT (MIN(CP_VALUE)
             FOR CP_NAME IN ('A', 'B', 'C','D','E')
            ) CUSTPIVOTED
```

An application wishing to find customers with a certain property can query the view directly, e.g.

```
SELECT * FROM EXTCUSTOMER
WHERE A IS NOT NULL
```

Figure 19 shows the performance obtained on the abstraction provided by the view. It compares three techniques:

- Store all the information in a single "wide" table that has five columns for the properties above. Have a single-column index of each of the properties.
- Have a separate property table, but do not exploit reordering properties, i.e. execute the view EXTCUSTOMER first and then apply additional operations such as filtering.
- Have a separate property table and enable PIVOT reordering.

To change the selectivity of predicates, we use different distributions for the property values. There are only 10 customers for which property 'A' is defined (i.e. not null in the "wide" row); then there are 100, 1000, 10000, and 100000 customers for which properties 'B' through 'E' are defined, respectively. For this experiment, we scaled up the number of customers from 150,000 to 600,000. Figure 20 shows the execution plan picked for properties 'A' and 'B' in this example, which are both very selective. An index seek is done against the non-clustered index on the CUSTPROPERTY table to determine what customers have this property. Then, another index seek is performed on clustered index of the CUSTOMER table to retrieve all the columns from the base table. Next, an index seek is performed to retrieve the remaining property values for this particular customer. After all the data has been assembled, it is sorted and stream aggregation is used to complete the pivot.

The property table is relatively small compared to the CUSTOMER table, so the performance difference between a separate property table and the "wide" table, when all the data is retrieved, is mostly due to the execution cost of our current PIVOT implementation. When there are predicates, our transformation rules can generate very efficient execution plans that exploit indices to locate qualifying rows quickly, making performance comparable to that obtained if we had a single table.

There is one restriction to point out regarding the benefits that can be obtained through indexing. When modeling directly as a "wide" table, it is possible to create and exploit multi-column indices. A separate property table does not naturally allow setting up such access paths, so the expected behavior is similar to that obtained with single-column indices.

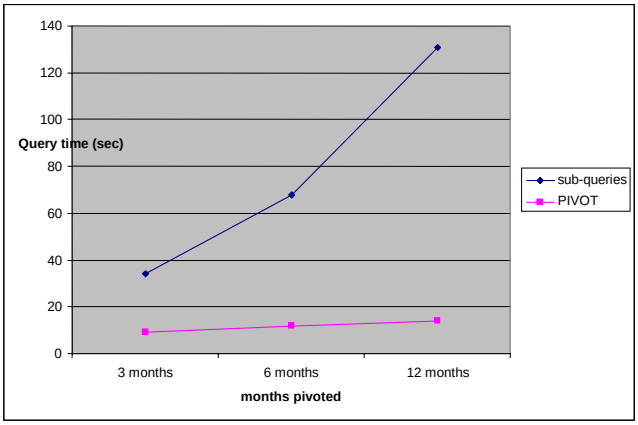



**Figure 18 PIVOT vs. scalar sub-queries**

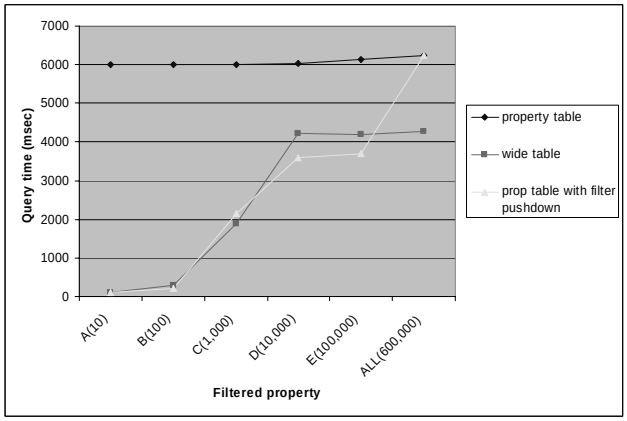



**Figure 19 PIVOT Property Table Results**

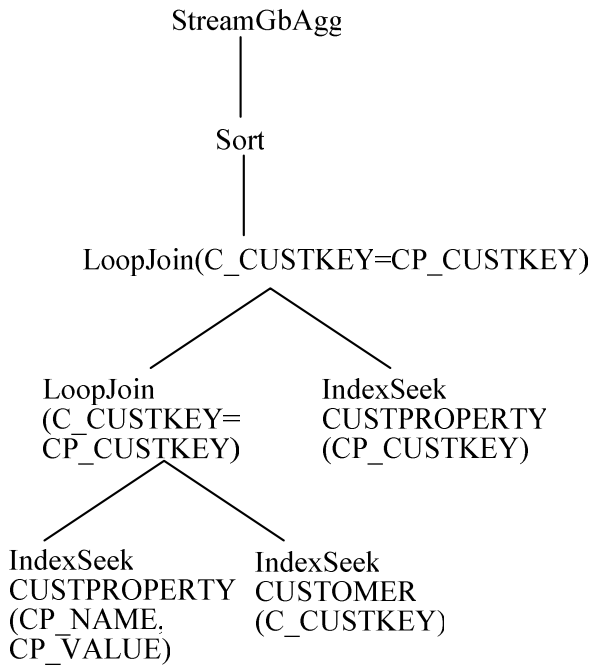



**Figure 20 Property Table With Filter Pushdown Plan**

## 6. Extensions

While this paper's PIVOT and UNPIVOT examples show a single value column, it is not difficult to extend this to an arbitrary number of columns. Each value column could be transposed into independent sets of columns in PIVOT, and UNPIVOT can similarly collapse different groups at the same time. Each group could use a different collapsing function (aggregate), and different aggregates could be used over the same column. Alternatively, complex data types could be created to hold multiple values in the same result column for as many values as are desired. None of these extensions significantly change the possible optimization or implementation techniques presented in this paper.

Extending the collapsing function also increases the utility of these two operators. The collapsing function is described as a single column aggregate function (SUM(), MIN(), etc.). Any RDBMS aggregate function, including user-defined or order-sensitive aggregate functions, could be allowed without affecting the transformations presented in this paper. Even more exotic aggregate functions, such as those that allow multiple input columns and/or produce multiple output columns, also fit nicely with simple extensions to the syntax proposed. Finally, it is possible to consider non-aggregate functions in this context. These could be used to throw an error when a data collision is detected in a cell or to handle data collisions by storing data from multiple rows as a nested relation or some other format that UNPIVOT can reassemble into multiple rows without losing data. These extensions allow a great deal of flexibility beyond traditional aggregation for PIVOT and UNPIVOT.

PIVOT and UNPIVOT are related to OLAP structures such as data cubes. However, OLAP operations do not always fit nicely into the SQL language. If a multi-dimensional structure is accessible through SQL, PIVOT and UNPIVOT could work on a portion of a cube visible as a relation (i.e. a two-dimensional set of rows and columns). Unpivoting a portion of a cube would be analogous to the operations presented for UNPIVOT in this paper.

## 7. Related Work

The idea behind PIVOT is not new. [4] described a number of extensions to grouping including a "cross-tab" query, though discussion was limited to PIVOT and did not discuss how to efficiently implement it or to expose pivoting below other operators.

SchemaSQL [5] implements transposing operations. The implementation appeared to be outside the RDBMS, however, and there was not significant discussion of query optimization in this context. [6] implements pivoting and unpivoting through unfold and fold operations, respectively. This work also does not attempt to push this capability deeply into the RDBMS.

[8] describes a system to expose spreadsheet-like functionality into a RDBMS, including how queries can be optimized using this approach. This model exposes behavior closer to OLAP than traditional flat relations, though some predicate pushing would be related in these two models.

[8] also describes a model of data n-dimensional array where cells are described through a coordinate system

using names. While similar in capabilities, the paradigm presented to the user differs from traditional SQL operators and may be harder to understand. We feel that our representation is a more natural representation of data rotation in the SQL language.

Both [8] and [4] describe that a relational table is a two-dimensional view of a cube, and this can represent a cross-tabulation of data. However, only [4] discusses how to move between the cross-tabulated and flat (narrow) form of data, and [4] only mentions that this capability exists in Microsoft Access.

Finally, our own prior work [2] exploited the basic design of unpivot operations for a special purpose that would be better served by deep integration into a database query processor.

## 8. Conclusion

We introduce two new data manipulation operators, Pivot and Unpivot, for use inside the RDBMS. These improve many existing user scenarios and enable several new ones. Furthermore, this paper outlines the basic syntactic, semantic, and implementation issues necessary to add this functionality to an existing RDBMS based on algebraic, cost-based optimization and algebraic data flow execution. Pivot is an extension of Group By with unique restrictions and optimization opportunities, and this makes it very easy to introduce incrementally on top of existing grouping implementations. Finally, we present a number of axioms of algebraic transformations useful in an implementation of Pivot and Unpivot.

## 8. References


[1] C. A. Galindo-Legaria, M. M. Joshi. Orthogonal Optimization of Subqueries and Aggregation, ACM SIGMOD 2001, May 21-24, 2001, Santa Barbara, California, USA, pages 571-581.

[2] G. Graefe, U. Fayyad, and S. Chaudhuri. On the Efficient Gathering of Sufficient Statistics for Classification from Large SQL Databases, Proceedings of The Fourth International Conference on Knowledge Discovery and Data Mining, 1998, pages 204-208.

[3] G. Graefe. The Cascades Framework for Query Optimization. Data Engineering Bulletin 18 (3) 1995, pages 19-29.

[4] J. Gray, A. Bosworth, A. Layman, H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, Data Mining and Knowledge Discovery, vol. 1, no. 1, 1997.

[5] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On Efficiently Implementing SchemaSQL on a SQL Database System. In Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, pages 471-482.

[6] R. Agrawal, A. Somani, Y. Xu. Storage and Querying of E-Commerce Data, In Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy, pages 149-158.

[7] M. Jaedicke, B. Mitshcang. On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS. 1998 Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, WA, pages 379-389.

[8] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Shen, S. Subramanian. Spreadsheets in RDBMS for OLAP. 2003 Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, CA, pages 52-63.