# Analytic SQL for Data Validation and Wrangling

SQL - A Flexible and Comprehensive Framework for In-Database Analytics

ORACLE WHITE PAPER | NOVEMBER 2016

## Contents

## Disclaimer

## Overview

Analytics is a must-have component of every corporate data warehouse and big data project. Many of the projects are trying to deliver analysis within their data reservoirs through the use of specialized languages and tools. This approach is locking data inside proprietary data silos, making cross-functional and cross-data store analysis either extremely difficult or completely impossible. IT teams are struggling to adapt complex silo-specific program code to support new and evolving analytical requirements. As a result, project costs are increasing and levels of risk within each project are also rising.

Many companies are searching for a single rich, robust, productive, standards driven language that can provide unified access over all types of data, drive rich sophisticated analysis, help constrain project costs and lower overall risk.

The flexibility and power of SQL makes it a vital tool for all data analysis projects and an ever-growing number of IT teams are using SQL as the go-to language for analysis. Already, many companies are using Oracle and SQL to drive sophisticated analysis across all their data sets as part of an agile development ecosystem.

The objective of this paper is to explain why SQL is fast becoming the default language for data validation and wrangling and how Oracle Database provides a rich, mature and comprehensive set of features and functions to support these use cases that avoids the continuous movement of data across different processing engines.

## SQL – A Flexible and Comprehensive Analytical Framework

The process of analyzing data has seen many changes and significant technological advances over the last forty years. However, there has been one language, one capability that has endured and evolved: the Structured Query Language or SQL. Many other languages and technologies have come and gone but SQL has been a constant. In fact, SQL has not only been a constant, but it has also improved significantly over time.

SQL is now the default language for data analytics because it provides a mature and comprehensive framework for data access and it supports a broad range of sophisticated analytical features. The key benefits for IT and business teams provided by Oracle's in-database analytical SQL features and functions are:

### Enhanced developer productivity

Using the latest built-in analytical SQL capabilities, developers can simplify their application code by replacing complex analytical processing – written using many different languages - with purpose-built analytical SQL that is much clearer and more concise. Tasks that in the past required the use of procedural languages or multiple SQL statements can now be expressed using single, comprehensive SQL statements. This simplified SQL (*analytic SQL*) is quicker to formulate, maintain and deploy compared to older approaches, resulting in greater developer productivity.

### Improved Manageability

When computations are centralized close to the data then the inconsistency, lack of timeliness and poor security of calculations scattered across multiple specialized processing platforms completely disappears. The ability to access a consolidated view of all your data is simplified when applications share a common relational environment rather than a mix of calculation engines with incompatible data structures and languages.

Oracle's in-database approach to analytics allows developers to efficiently layer their analysis using SQL because it can support a very broad range of business requirements.

### Minimized Learning Effort

The amount of effort required to understand analytic SQL is minimized through the use of careful syntax design. Syntax typically leveraged existing SQL constructs, such as the aggregate functions SUM and AVG, and extends them using well-understood keywords such as `OVER`, `PARTITION BY`, `ORDER BY`, `RANGE INTERVAL` etc.

Most developers and business users with a reasonable level of proficiency with SQL and can quickly adopt and integrate sophisticated analytical features, such as pareto-distributions, pattern matching, cube and rollup aggregations into their applications and reports.

The amount of time required for enhancements, maintenance and upgrades is minimized: more people will be able to review and enhance the existing SQL code rather than having to rely on a few key people with specialized programming skills.

### ANSI SQL compliance

Most of Oracle's analytical SQL is part of the ANSI SQL standard; or in the process of becoming adopted in newer versions. This ensures broad support for these features and rapid adoption of newly introduced functionality across applications and tools – both from Oracle's partner network and other independent software vendors.

Oracle is continuously working with its many partners to assist them in exploiting the expanding library of analytic functions. Already many independent software vendors have integrated support for the new Database 12c in-database analytic functions into their products.

### Improved performance

Oracle's in-database analytical functions and features enable significantly better query performance. Not only does it remove the need for specialized data-processing silos but also the internal processing of these purpose-built functions is fully optimized. Using SQL unlocks the full potential of the Oracle database - such as parallel execution – to provide enterprise level scalability unmatched by external specialized processing engines.

### Summary

This section has outlined how Oracle's in-database analytic SQL features provide IT, application development teams and business users with a robust and agile analytical language that enhances both query performance and productivity while providing investment protection by building on existing standards-based skills. For a more detailed analysis of the benefits of SQL as an analysis language please refer to the following whitepaper: SQL – the natural language for analysis

The rest of this paper will outline the key SQL-based features for data transformation and data wrangling within Oracle Database 12c Release 2 [1].

## Analytical Transformation Features

### Simplified validation of data types

Many companies are investing heavily in acquiring new external data sources. Sometimes the data within these external sources is inexact. Consequently conversion errors happen either during data loading jobs or when running reports. In the past these conversion errors would typically result in a job or query aborting. Developers can, and do, invest considerable time and effort creating workarounds and implementing code to overcome data type conversion errors.

Many data integration workflows fail at some point in time because of conversion errors. With the growing interest in big data many IT teams are working with more and more external data sources. Many of these tend to contain inexact data, which typically results in conversion errors.

To avoid these errors many ETL code generators create lots of additional code to prevent data load processes aborting when a conversion error occurs. This additional error-checking code can have a significant performance impact.

It is not uncommon for complex data models to contain tables where multiple different foreign keys are stored in one common column. This is becoming increasingly common where there is a need to support integration with external data sources.  These multiple foreign keys may have different data types, therefore, the "key" column is often declared as VARCHAR2. For some rows the value might be a numeric foreign key to table A, and for other rows it might be a VARCHAR2 foreign key to table B.

In simple terms, developers and DBAs need standard SQL syntax that will convert bad data points to a predefined default value, avoiding the generation of errors and possibly aborting a process.

Database 12c Release 2 has extended the existing SQL `CAST` function so that it returns a user-specified value if there is a conversion error. In addition, a new SQL function, `VALIDATE_CONVERSION`, can be used to determines whether a given input can be converted to a given data type. This function will help identify problem data that cannot be converted.

1 Oracle Database 12c Release 2 (12.2), the latest generation of the world's most popular database, is now available in the Oracle Cloud

## Code Sample

The SQL CAST function now has the ability to return a user-specified value if there is a conversion error. For example, when processing a new data source for the first time, it would be prudent to check the validity of the data types across the source column using something similar to the following code:

```
SELECT
 CAST(e.empno AS NUMBER) AS empno,
 e.ename,
 CAST(e.hiredate AS date) AS hiredate,
 e.deptno
FROM my_emp e ;
```

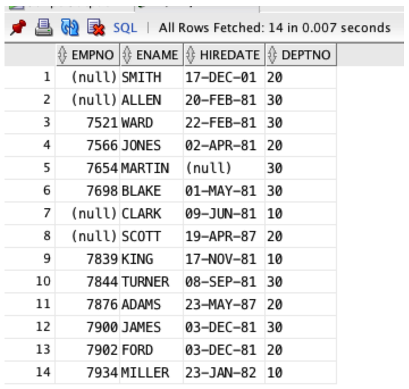
Prior to latest release of the Oracle Database, any non-numeric values in the column would trigger the following error:

```
ORA-01722: invalid number
01722. 00000 -  "invalid number"
*Cause:    The specified number was invalid.
*Action:   Specify a valid number.
```

Note that the error does not indicate which values caused the error. With 12c Release 2 the above error can be avoided using the new ‘`ON CONVERSION ERROR`’ syntax as shown below.

```
SELECT
 CAST(e.empno AS NUMBER DEFAULT NULL ON CONVERSION ERROR) AS empno,
 e.ename,
 CAST(e.hiredate AS date DEFAULT NULL ON CONVERSION ERROR) AS hiredate,
 e.deptno
FROM my_emp e;
```

This new syntax returns the following result where rows 1, 2 and 5 contain the value *null* in the empno and hiredate columns indicating that at conversion error has been caught:



| | EMPNO | ENAME | HIREDATE | DEPTNO |
|---|---|---|---|---|
| 1 | (null) | SMITH | 17-DEC-01 | 20 |
| 2 | (null) | ALLEN | 20-FEB-81 | 30 |
| 3 | 7521 | WARD | 22-FEB-81 | 30 |
| 4 | 7566 | JONES | 02-APR-81 | 20 |
| 5 | 7654 | MARTIN | (null) | 30 |
| 6 | 7698 | BLAKE | 01-MAY-81 | 30 |
| 7 | (null) | CLARK | 09-JUN-81 | 10 |
| 8 | (null) | SCOTT | 19-APR-87 | 20 |
| 9 | 7839 | KING | 17-NOV-81 | 10 |
| 10 | 7844 | TURNER | 08-SEP-81 | 30 |
| 11 | 7876 | ADAMS | 23-MAY-87 | 20 |
| 12 | 7900 | JAMES | 03-DEC-81 | 30 |
| 13 | 7902 | FORD | 03-DEC-81 | 20 |
| 14 | 7934 | MILLER | 23-JAN-82 | 10 |

FIGURE 1 – SAMPLE DATASET WITH ERRORS IN ROWS 1, 2 AND 5

However, it is important to note that in this specific example, a *bad* number will now result in a missing row within the target table and not an “`ORA-01722: invalid number`”.

The following TO_xxx conversion functions have also been extended to return a user-specified value if there is a conversion error:

- TO_NUMBER
- TO_BINARY_FLOAT/TO_BINARY_DOUBLE
- TO_DATE
- TO_TIMESTAMP/TO_TIMESTAMP_TZ
- TO_DSINTERVAL/TO_YM_INTERVAL

These new capabilities mean that both DBAs and developers can now simplify existing code, making it easier to read and easier to maintain.

A new function VALIDATE_CONVERSION() has been added to make it easier to identify rows in a table or view where a conversion failure will occur. This function returns 1 if the expression can be converted to the specified data type otherwise it returns 0. Extending the previous code example, the following query returns all rows from the table s where the column s.vc fails to convert the data to a number:

```
SELECT * FROM s
WHERE VALIDATE_CONVERSION(s.vc AS NUMBER) = 0;
```

These new features can be used to reduce the complexity of data validation code and error trapping within reports with calls to native SQL functions. This results in better performance of jobs and queries and reduced overall system resource usage.

## Simpler management of large aggregated text lists

The most widely used method for creating lists of values within a result set is the LISTAGG function. The key challenge when using LISTAGG has been the possible runtime overflow of the function due to a string being too long - which then generates an error and aborts the query and/or workflow.

With Database 12c Release 2 the LISTAGG functionality has been enhanced. It now provides a way to truncate the string to fit within the limit of the VARCHAR2 object [2]. Specific rules are used to correctly manage the truncation of complete words and determine how many values have been truncated. It is possible to provide an overflow/truncation identifier character string as part of the definition.

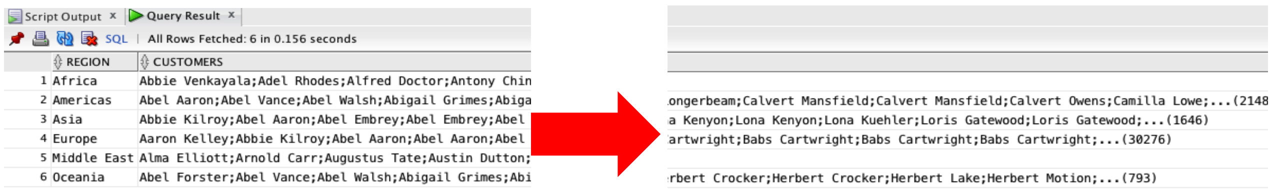
### Code Sample

Using the new overflow functionality the query below, which would have resulted in an error prior to 12.2, now succeeds as shown:

```
SELECT
  g.country_region,
  LISTAGG(c.cust_first_name||' '||c.cust_last_name, ',' ON OVERFLOW TRUNCATE '...'
WITH COUNT) WITHIN GROUP (ORDER BY c.country_id) AS Customer
FROM customers c, countries g
WHERE g.country_id = c.country_id
GROUP BY country_region
ORDER BY country_region;
```

2 With Database 12c the size of a VARCHAR2 object has been increased to 32K. This is controlled by the database setting MAX_STRING_SIZE. For more information please refer to the Database Reference Guide

This returns a result similar to the following (*to show the new count of missing values the image of the output window from SQL Developer has been split.*):



```
Script Output ×   Query Result ×
SQL | All Rows Fetched: 6 in 0.156 seconds
    REGION      CUSTOMERS
  1 Africa      Abbie Venkayala;Adel Rhodes;Alfred Doctor;Antony Chin
  2 Americas    Abel Aaron;Abel Vance;Abel Walsh;Abigail Grimes;Abiga      ongerbeam;Calvert Mansfield;Calvert Mansfield;Calvert Owens;Camilla Lowe;...(2148
  3 Asia        Abbie Kilroy;Abel Aaron;Abel Embrey;Abel Embrey;Abel       a Kenyon;Lona Kenyon;Lona Kuehler;Loris Gatewood;Loris Gatewood;...(1646)
  4 Europe      Aaron Kelley;Abbie Kilroy;Abel Aaron;Abel Aaron;Abel       artwright;Babs Cartwright;Babs Cartwright;Babs Cartwright;...(30276)
  5 Middle East Alma Elliott;Arnold Carr;Augustus Tate;Austin Dutton;
  6 Oceania     Abel Forster;Abel Vance;Abel Walsh;Abigail Grimes;Abi      rbert Crocker;Herbert Crocker;Herbert Lake;Herbert Motion;...(793)
```

FIGURE 2 – AN EXAMPLE OF NEW LISTAGG FEATURE TO MANAGE LONG STRINGS – SHOWING COUNT OF MISSING VALUES IN BRACKETS

The characters to indicate that an overflow has occurred are appended at the end of the list of values, which in this case if the default value of three dots "`. . .`". The overflow functionality traverses backwards from the maximum possible length to the end of the last complete value in the LISTAGG clause, then it adds the user-defined separator followed by the user defined overflow indicator, followed by output from the `'WITH COUNT'` clause which adds a counter at the end of a truncated string to indicate the number of values that have been removed/truncated from the list.

## Analytical Data Wrangling Functions

### Pivoting operations

Pivoting is a key technique in many data warehouse related applications. Business intelligence queries typically transform multiple input rows into result sets that contain fewer and generally wider rows. The data returned by business intelligence queries is often most usable if presented in a cross-tabular format. The pivot clause of the SELECT statement lets application developers write cross-tabular queries that rotate rows into columns, aggregating data as part of the rotation process. Prior to this feature BI tools and report developers were forced to use self-joins as a workaround, which resulted in complex code that was extremely difficult to optimize.

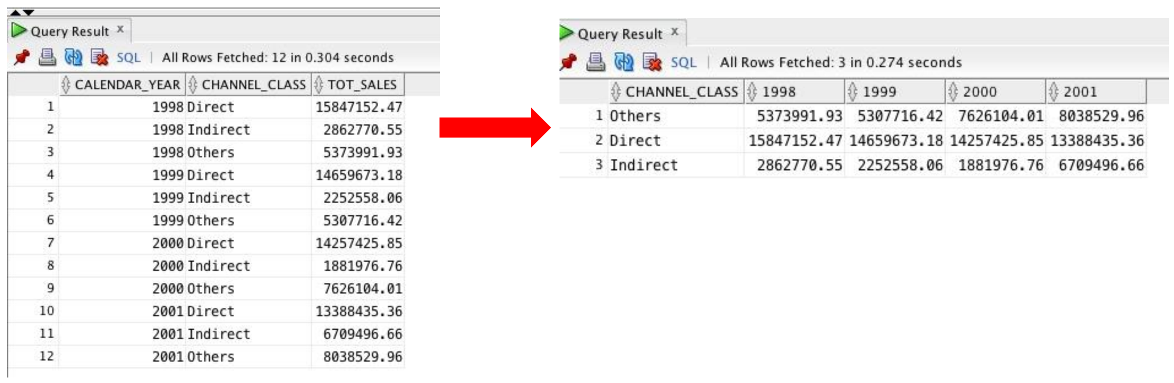
Many ETL and data mining workflows include pivot operations within the transformation step. Pushing this type of processing back inside the database and executing it using native SQL will significantly the increase performance of workflows that require this type of feature by removing the need to deploy staging tables to manage the row-to-column or column-to-row transformation.

#### Code Sample

If we need to convert dimension values (rows) into columns to split out a specific set of data points then we can use the PIVOT clause to transform the result set. For example, using Oracle's SQL PIVOT functionality transactional data can be easily transformed and represented as dimensional aggregated information using the following SQL:

```
SELECT * FROM
(SELECT
 t.calendar_year,
 c.channel_class,
 s.amount_sold as tot_sales
FROM sales s, times t, channels c
WHERE s.time_id = t.time_id
AND s.channel_id = c.channel_id)
PIVOT
(SUM(tot_sales) FOR calendar_year IN (1998, 1999, 2000, 2001));
```

which returns the following output:


Query Result
All Rows Fetched: 12 in 0.304 seconds

| | CALENDAR_YEAR | CHANNEL_CLASS | TOT_SALES |
|---|---|---|---|
| 1 | 1998 | Direct | 15847152.47 |
| 2 | 1998 | Indirect | 2862770.55 |
| 3 | 1998 | Others | 5373991.93 |
| 4 | 1999 | Direct | 14659673.18 |
| 5 | 1999 | Indirect | 2252558.06 |
| 6 | 1999 | Others | 5307716.42 |
| 7 | 2000 | Direct | 14257425.85 |
| 8 | 2000 | Indirect | 1881976.76 |
| 9 | 2000 | Others | 7626104.01 |
| 10 | 2001 | Direct | 13388435.36 |
| 11 | 2001 | Indirect | 6709496.66 |
| 12 | 2001 | Others | 8038529.96 |

Query Result
All Rows Fetched: 3 in 0.274 seconds

| | CHANNEL_CLASS | 1998 | 1999 | 2000 | 2001 |
|---|---|---|---|---|---|
| 1 | Others | 5373991.93 | 5307716.42 | 7626104.01 | 8038529.96 |
| 2 | Direct | 15847152.47 | 14659673.18 | 14257425.85 | 13388435.36 |
| 3 | Indirect | 2862770.55 | 2252558.06 | 1881976.76 | 6709496.66 |

FIGURE 3 – AN EXAMPLE OF PIVOT FUNCTION – ROTATES ROWS TO COLUMNS

### Implicit Data Aggregation

The above example does not include an explicit data aggregation method or a `GROUP BY` clause within the inner SELECT statement:

```
 s.amount_sold as tot_sales
FROM sales s, times t, channels c
WHERE. . .)
```

The PIVOT function applies the grouping and aggregation behind the scenes. In the above example the PIVOT operation enforces a SUM() operator on the `tot_sales` column.

```
PIVOT
(SUM(tot_sales) FOR
```

The SUM operator can be replaced by any of the normal aggregation functions that are supported for GROUP BY operations.

### Renaming Column Headings

It is possible to rename each of the column headings using an alias, as shown below:

```
PIVOT
(SUM(tot_sales)
  FOR calendar_year IN ('1998' AS YR_1998,
                        '1999' AS YR_1999,
                        '2000' AS YR_2000,
                        '2001' AS YR_2001));
```

Note that the IN (…) clause actually lists the row values that will be transposed to columns. There are two options when it is not possible to identify these values in advance:

1. Use XML+ANY keyword
2. Use the XML keyword + subquery

### ANY Keyword

The ANY keyword can only be used in conjunction with the PIVOT XML clause. The output densifies the data to include all possible time periods at the year-level for each channel.

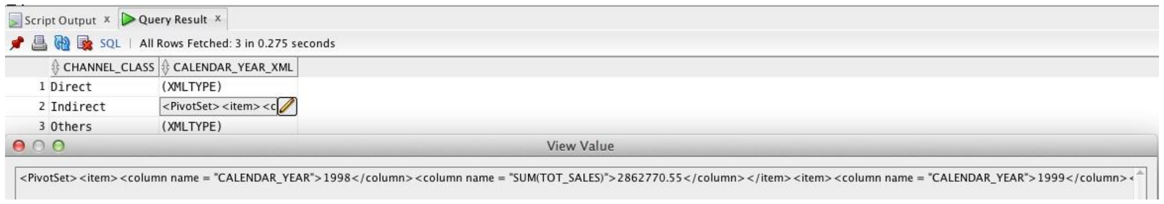
```
SELECT * FROM
(SELECT
 t.calendar_year,
 c.channel_class,
 s.amount_sold as tot_sales
FROM sales s, times t, channels c
WHERE s.time_id = t.time_id
AND s.channel_id = c.channel_id)
PIVOT XML (SUM(tot_sales) FOR calendar_year IN (ANY));
```

### XML+Subquery

The XML clause within the PIVOT function makes this possible:

```
SELECT * FROM
(SELECT
 t.calendar_year,
 c.channel_class,
 s.amount_sold as tot_sales
FROM sales s, times t, channels c
WHERE s.time_id = t.time_id
AND s.channel_id = c.channel_id)
PIVOT XML(SUM(tot_sales) FOR calendar_year IN (SELECT DISTINCT calendar_year
FROM times));
```

In this particular case both the ANY and subquery code samples generate the following output:



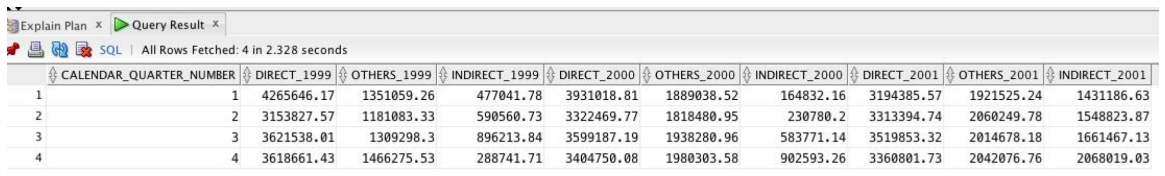
```
All Rows Fetched: 3 in 0.275 seconds

   CHANNEL_CLASS  CALENDAR_YEAR_XML
 1 Direct         (XMLTYPE)
 2 Indirect       <PivotSet><item><c
 3 Others         (XMLTYPE)

View Value
<PivotSet><item><column name = "CALENDAR_YEAR">1998</column><column name = "SUM(TOT_SALES)">2862770.55</column></item><item><column name = "CALENDAR_YEAR">1999</column><
```

FIGURE 4 – AN EXAMPLE OF PIVOT FUNCTION USING ANY/SUBQUERY AND RETURNING XML FORMATTED RESULTS

### Pivoting Multiple Columns

It is possible to pivot on more than one column. The example below shows a typical multiple column pivot operation using a multi-column IN-list with column headings designed to match the IN-list members. Column value combinations that are not included in the IN-list clause are simply omitted from the resultset:

```
SELECT * FROM
(SELECT
 t.calendar_year,
 t.calendar_quarter_number,
 c.channel_class,
 s.amount_sold as tot_sales
FROM sales s, times t, channels c
WHERE s.time_id = t.time_id
AND s.channel_id = c.channel_id)
PIVOT
(SUM(tot_sales) FOR (calendar_year, channel_class) IN
   ((1999, 'Direct') AS Direct_1999,
    (1999, 'Others') AS Others_1999,
    (1999, 'Indirect') AS Indirect_1999,
    (2000, 'Direct') AS Direct_2000,
    (2000, 'Others') AS Others_2000,
    (2000, 'Indirect') AS Indirect_2000,
    (2001, 'Direct') AS Direct_2001,
    (2001, 'Others') AS Others_2001,
    (2001, 'Indirect') AS Indirect_2001
   ))
order by 1;
```



Explain Plan | Query Result

SQL | All Rows Fetched: 4 in 2.328 seconds

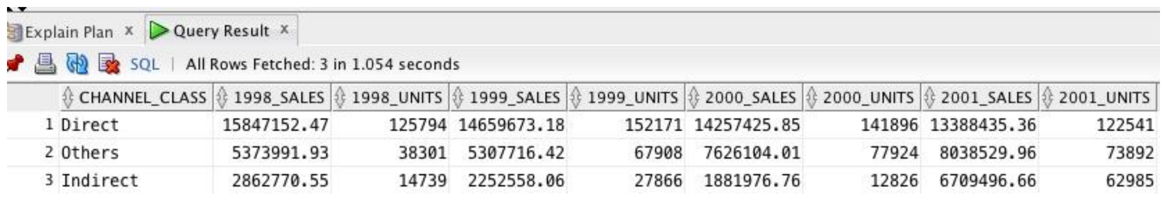
| | CALENDAR_QUARTER_NUMBER | DIRECT_1999 | OTHERS_1999 | INDIRECT_1999 | DIRECT_2000 | OTHERS_2000 | INDIRECT_2000 | DIRECT_2001 | OTHERS_2001 | INDIRECT_2001 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 4265646.17 | 1351059.26 | 477041.78 | 3931018.81 | 1889038.52 | 164832.16 | 3194385.57 | 1921525.24 | 1431186.63 |
| 2 | 2 | 3153827.57 | 1181083.33 | 590560.73 | 3322469.77 | 1818480.95 | 230780.2 | 3313394.74 | 2060249.78 | 1548823.87 |
| 3 | 3 | 3621538.01 | 1309298.3 | 896213.84 | 3599187.19 | 1938280.96 | 583771.14 | 3519853.32 | 2014678.18 | 1661467.13 |
| 4 | 4 | 3618661.43 | 1466275.53 | 288741.71 | 3404750.08 | 1980303.58 | 902593.26 | 3360801.73 | 2042076.76 | 2068019.03 |

FIGURE 5 – AN EXAMPLE OF PIVOTING MULTIPLE COLUMNS

## Pivoting Multiple Aggregates

The pivot clause also supports multiple aggregates as shown below:

```
SELECT * FROM
(SELECT
 t.calendar_year,
 c.channel_class,
 s.amount_sold as tot_sales,
 s.quantity_sold as tot_units
FROM sales s, times t, channels c
WHERE s.time_id = t.time_id
AND s.channel_id = c.channel_id
ORDER BY 1, 2)
PIVOT
(SUM(tot_sales) AS sales,
 SUM(tot_units) AS units
  FOR calendar_year IN (1998, 1999, 2000, 2001));
```



Explain Plan | Query Result

SQL | All Rows Fetched: 3 in 1.054 seconds

| | CHANNEL_CLASS | 1998_SALES | 1998_UNITS | 1999_SALES | 1999_UNITS | 2000_SALES | 2000_UNITS | 2001_SALES | 2001_UNITS |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Direct | 15847152.47 | 125794 | 14659673.18 | 152171 | 14257425.85 | 141896 | 13388435.36 | 122541 |
| 2 | Others | 5373991.93 | 38301 | 5307716.42 | 67908 | 7626104.01 | 77924 | 8038529.96 | 73892 |
| 3 | Indirect | 2862770.55 | 14739 | 2252558.06 | 27866 | 1881976.76 | 12826 | 6709496.66 | 62985 |

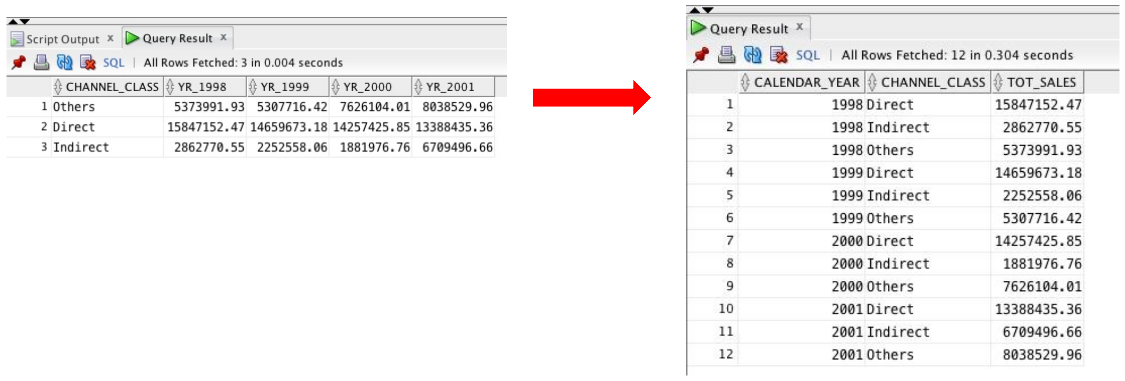
FIGURE 6 – AN EXAMPLE OF PIVOTING MULTIPLE AGGREGATIONS

## Unpivoting Operations

The UNPIVOT clause lets you rotate the data the other way – i.e. it is the reverse of pivot. For example:

```
SELECT calendar_year,
       channel_class,
       amount_sold
FROM pivoted_data
 UNPIVOT (amount_sold FOR calendar_year IN (YR_1998 AS '1998',
                                            YR_1999 AS '1999',
                                            YR_2000 AS '2000',
                                            YR_2001 AS '2001'))
ORDER BY 1,2;
```

which returns the following output:



Script Output × | Query Result ×

SQL | All Rows Fetched: 3 in 0.004 seconds

| | CHANNEL_CLASS | YR_1998 | YR_1999 | YR_2000 | YR_2001 |
|---|---|---|---|---|---|
| 1 | Others | 5373991.93 | 5307716.42 | 7626104.01 | 8038529.96 |
| 2 | Direct | 15847152.47 | 14659673.18 | 14257425.85 | 13388435.36 |
| 3 | Indirect | 2862770.55 | 2252558.06 | 1881976.76 | 6709496.66 |

Query Result ×

SQL | All Rows Fetched: 12 in 0.304 seconds

| | CALENDAR_YEAR | CHANNEL_CLASS | TOT_SALES |
|---|---|---|---|
| 1 | 1998 | Direct | 15847152.47 |
| 2 | 1998 | Indirect | 2862770.55 |
| 3 | 1998 | Others | 5373991.93 |
| 4 | 1999 | Direct | 14659673.18 |
| 5 | 1999 | Indirect | 2252558.06 |
| 6 | 1999 | Others | 5307716.42 |
| 7 | 2000 | Direct | 14257425.85 |
| 8 | 2000 | Indirect | 1881976.76 |
| 9 | 2000 | Others | 7626104.01 |
| 10 | 2001 | Direct | 13388435.36 |
| 11 | 2001 | Indirect | 6709496.66 |
| 12 | 2001 | Others | 8038529.96 |

FIGURE 6 – AN EXAMPLE OF UNPIVOT FUNCTION – ROTATES COLUMNS TO ROWS

## Conclusion

Oracle's analytical SQL features and functions provide business users, developers and DBAs with a comprehensive and powerful way to support the most data wrangling requirements within ETL workflows and reports. By moving this type of processing inside the Oracle Database DBAs and developers will benefit from increased productivity and business users will benefit from improved query performance across a broad range of use cases.

The use of Oracle's in-database data wrangling features and functions deliver the following benefits to IT teams and business users:

- Increased developer productivity
- Minimizes learning effort
- Improves manageability
- Provides investment protection (adheres to industry standards based syntax)
- Delivers increased query speed

The flexibility and power of Oracle's data wrangling features, combined with their adherence to international SQL standards, makes them an important tool for all SQL users.

Overall, the SQL analytic functions and features in Oracle Database 12c Release 2 make it the most effective platform for delivering analytical results directly into operational, data warehousing and business intelligence projects.

## Further Reading

See the following links for more information about the in-database analytic features that are part of Oracle Database:

1. Database SQL Language Reference - Oracle and Standard SQL
2. **Oracle Analytical SQL Features and Functions** - a compelling array of analytical features and functions accessible through SQL. Available via the Analytic SQL home page on OTN.
3. **SQL - the natural language for analysis** – a review of the reasons why SQL is the best language for data analysis. Available via the Analytic SQL home page on OTN.
4. **Oracle Statistical Functions** - eliminate movement and staging to external systems to perform statistical analysis. For more information see the SQL Statistical Functions home page on OTN.
5. Oracle Database 12c Query Optimization - providing innovation in plan execution and stability.

The following Oracle whitepapers, articles, presentations and data sheets are essential reading and available via the Analytic SQL home page on OTN:

   a. SQL for Data Validation and Data Wrangling
   b. SQL for Analysis, Reporting and Modeling
   c. SQL for Advanced Data Aggregation
   d. SQL for Approximate Query Processing
   e. SQL for Pattern Matching

2. Oracle Magazine SQL 101 Columns
3. Oracle Database SQL Language Reference—T-test Statistical Functions
4. Oracle Statistical Functions Overview
5. SQL Analytics Data Sheet

You will find links to the above papers, and more, on the "Oracle Analytical SQL" web page hosted on the Oracle Technology Network:

http://www.oracle.com/technetwork/database/bi-datawarehousing/sql-analytics-index-1984365.html