String Aggregation Techniques in Oracle ListAgg – Details, Limitations, and Alternatives

Cristian DUMITRESCU
IBM Romania
mail.cristian.dumitrescu@gmail.com

This paper proposes several ways to overcome the ORA-01489 result of string concatenation is too long error in early versions of Oracle Database (before 12c). Each proposed alternative is presented with pros and cons and may be applied only in specific cases, depending on the requirement or cause.
Keywords: *Relational Databases, Oracle, ListAgg, ORA-01489, User-Defined Functions*

1 Introduction

When working with databases, the aggregation of character fields is a common need in the development of views and reports.
To meet these requirements, Oracle provides a convenient solution through the *ListAgg* function.
As the lifetime support for Oracle 11g has ended in 2020, most companies are looking to migrate to a newer version [1]. However, the 11g version is still widely used.
The paper aims to summarize the options that the database developer has at its disposal to overcome the 4000-character limit imposed on the *ListAgg* function in the Oracle 11g version.

2. The evolution of the built-in *ListAgg* function

ListAgg was first introduced in the Oracle world in version 11gR2.
ListAgg aggregates the values of multiple rows into a single grouping. The rows are, thereby, “denormalized” in a single concatenation of values, optionally delimited by a comma, thus generating a csv (comma-separated-value) result. Similar to other built-in Oracle functions, *ListAgg* can be used as an aggregate function (along with the “group by” clause), or as an analytical function (along with the “*over*” and “*partition by*” clauses).
ListAgg is most often used in conjunction with character attributes, although the function can receive other data types as input.
Another favorable feature of the function is the ability to sort concatenated elements within a group.

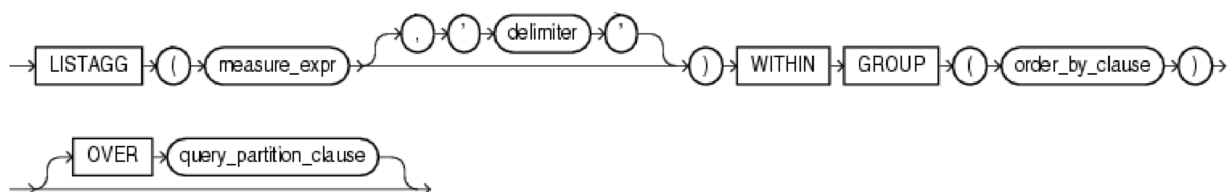


Fig. 1. *ListAgg* syntax [2]

Note: All SQL queries used in this paper will run on Oracle’s well-known HR schema.

```
select r.region_name,
       LISTAGG(c.country_name,'; ')  WITHIN GROUP (ORDER BY c.country_name)
region_countries
  from regions r
  join countries c on r.region_id = c.region_id
 group by r.region_name;
```

	REGION_NAME	REGION_COUNTRIES
1	Americas	Argentina; Brazil; Canada; Mexico; United States of America
2	Asia	Australia; China; India; Japan; Malaysia; Singapore
3	Europe	Belgium; Denmark; France; Germany; Italy; Netherlands; Switzerland; United Kingdom
4	Middle East and Africa	Egypt; Israel; Kuwait; Nigeria; Zambia; Zimbabwe

Fig. 2. *ListAgg* example in aggregate mode

```
select c.*,
      LISTAGG(c.country_id,';')  WITHIN GROUP (order by null) over
(partition by region_id ) COUNTRIES_IN_REGION
 from countries c
where c.region_id = 2;
```

	COUNTRY_ID	COUNTRY_NAME	REGION_ID	COUNTRIES_IN_REGION
1	US	United States of America	2	US;CA;BR;MX;AR
2	CA	Canada	2	US;CA;BR;MX;AR
3	BR	Brazil	2	US;CA;BR;MX;AR
4	MX	Mexico	2	US;CA;BR;MX;AR
5	AR	Argentina	2	US;CA;BR;MX;AR

Fig. 3. *ListAgg* example in analytical mode

The optional delimiter should be carefully chosen because in the output it can be confused with a part of the field being concatenated. Best practice dictates choosing a special character that is not present in the aggregated fields.
The result of the function is a *varchar2*, limited to 4000 characters.

Before the 11g version, Oracle had no direct mechanism that allowed multiple values of the same column to be displayed on a single row in the output. In a post on his blog, Donald Burlescon presents some solutions to this problem: using the *XMLAgg* function (Oracle 9i) and using the *SYS_CONNECT_BY_PATH* operator [3], [4].

```
select r.region_name
      ,rtrim(xmlagg(xmlelement(e, c.country_name || ';') ORDER BY
c.country_name).extract ('//text()'), ';') region_countries_XMLAGG
  from regions r
  join countries c on r.region_id = c.region_id
 group by r.region_name;
```

	REGION_NAME	REGION_COUNTRIES_XMLAGG
1	Americas	Argentina;Brazil;Canada;Mexico;United States of America
2	Asia	Australia;China;India;Japan;Malaysia;Singapore
3	Europe	Belgium;Denmark;France;Germany;Italy;Netherlands;Switzerland;United Kingdom
4	Middle East and Africa	Egypt;Israel;Kuwait;Nigeria;Zambia;Zimbabwe

Fig. 4. *ListAgg* alternative – String concatenation with *XMLAgg*

```
select region_name,
        substr(SYS_CONNECT_BY_PATH(country_name, ','),2) name_list
```

```
   from (select r.region_name,
                c.country_name,
                count(*) OVER ( partition by r.region_name ) cnt,
                ROW_NUMBER () OVER ( partition by r.region_name order by
c.country_name) seq
           from regions r
           join countries c on r.region_id = c.region_id          where
r.region_name is not null)
  where seq = cnt
  start with seq = 1
connect by  prior seq + 1 = seq
        and prior region_name = region_name;
```

	REGION_NAME	NAME_LIST
1	Americas	Argentina,Brazil,Canada,Mexico,United States of America
2	Asia	Australia,China,India,Japan,Malaysia,Singapore
3	Europe	Belgium,Denmark,France,Germany,Italy,Netherlands,Switzerland,United Kingdom
4	Middle East and Africa	Egypt,Israel,Kuwait,Nigeria,Zambia,Zimbabwe

Fig. 5. *ListAgg* alternative – String concatenation with *SYS_CONNECT_BY_PATH*

In the event that the function exceeds the 4000 characters limit, the following runtime error occurs:

```
ORA-01489: result of string
concatenation is too long
```

In Oracle 12c, *ListAgg* has received an upgrade through which this limitation can be elegantly overcome - the *ON OVERFLOW* clause.

```
  with my_query as (select level no
                      from dual connect by level <10000)
select LISTAGG(no,'; ' on overflow truncate) WITHIN GROUP (order by null)
as no_list
  from my_query;
```

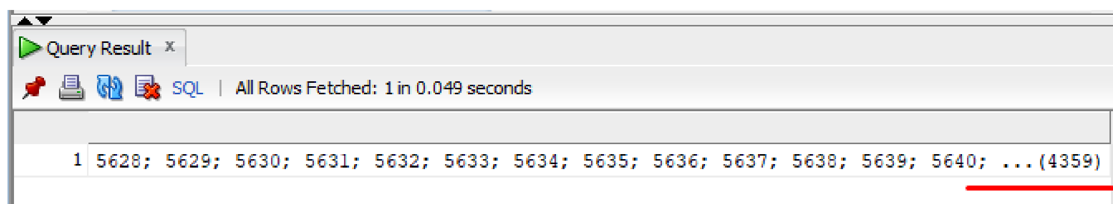


Fig. 6. *ListAgg* with *ON OVERFLOW* clause

The query above would generate an error if we deleted or commented on the "*on overflow truncate*" clause.
If the string aggregation output exceeds 4000 characters, the function truncates the result before this threshold, notifying the user with a customizable message (the example above uses the standard option of "...") [5].
Starting with the Oracle 19c version, the function receives new improvements: The "within group" clause becomes optional, and the concatenation can be performed for distinct values, by using *ListAgg* and *DISTINCT* together [6].

3. Avoiding *ListAgg's* Overflow Error

Runtime Errors such as ORA-01489 are difficult to detect during development because the syntax itself is correct, the error is actually caused by the volume of the data. For example, a view that uses this function can be developed in a test environment with a small volume of data and will not generate this error until it is introduced in the production database. Deleting or restricting

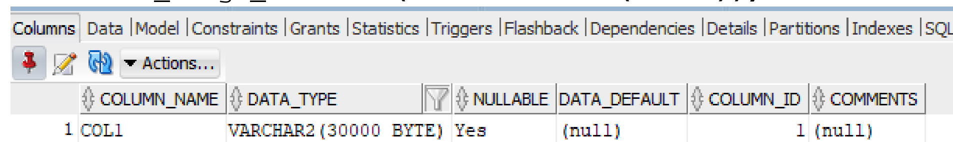
the data so that the function falls within the range dictated by the threshold is certainly not a solution. Several options will be explored, along with pros and cons depending on how *ListAgg* is used.
Newer versions of Oracle provide multiple solutions to this common problem:

3.1 Replacing ListAgg with XMLAgg

By performing this operation, the view in question will no longer produce a runtime error. This solution should be applied with caution, as the data type of the result changes from *varchar2* to *clob*. If the view is subsequently used by various reporting or ETL tools, they may reject such a field.
In situations where returning the entire result is mandatory, you can choose this option.

3.2 Extending the varchar2 limit from 4000 up to 32767 characters, by setting *MAX_STRING_SIZE* initialization parameter to *EXTENDED*.

```
create table test_large_varchar (col1 varchar2(30000));
```



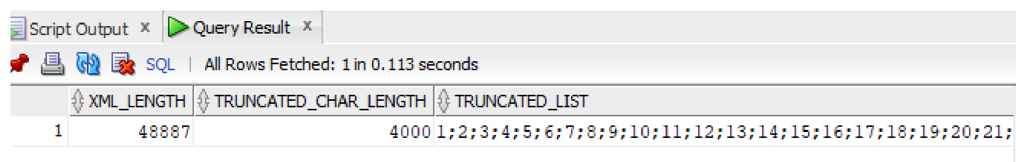
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	COL1	VARCHAR2(30000 BYTE)	Yes	(null)	1	(null)

Fig. 7. Creating a table with a *varchar2* field of more than 4000 characters

Admin rights are required to apply this method. The database must also be in *Upgrade* mode.
This option does not directly solve the current issue, but it can prove to be a decent solution if the developer needs to store such aggregation in a table within a non-clob field.

3.3 Replacing ListAgg with Substring, To_Char, XML_Agg

```
  with my_query as (select level no from dual connect by level <10000)
select length(t.no_list) xml_length,
       length(to_char(substr(t.no_list,1,4000))) truncated_char_length,
       to_char(substr(t.no_list,1,4000)) truncated_list
from (
select rtrim(xmlagg(xmlelement(e, no || ';') ORDER BY null).extract
('//text()').getclobval(), ';') as no_list
  from my_query) t;
```



	XML_LENGTH	TRUNCATED_CHAR_LENGTH	TRUNCATED_LIST
1	48887	4000	1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20;21;

Fig. 8. Replicating *on overflow truncate* clause of *ListAgg* by using a combination of *Substring*, *To_Char,* and *XML_Agg*

With this method, the output is first truncated if the string aggregation exceeds 4000 characters. The result is then transformed from *clob* to character data type.

Note: *XMLAgg* can return the result in both *clob* or character data types, using *getclobval()* or *getstringval()*. In the above query, using *getstringval()* is not an option because it would generate an error similar to the one we were trying to avoid.
This option can be chosen in situations where keeping the character data type prevails over a complete string aggregation output.

3.4 Creating new User-Defined Aggregate Functions

Starting with Oracle 9i, developers can create custom Aggregate Functions. Keith Laker, Oracle's Senior Principal Product Manager, details such an example of a user-defined aggregate function in his blog post [7]. The same topic is also addressed on the famous IT blogs https://www.stackoverflow.com and AskTom [8].

According to Oracle [9], User-defined aggregate functions are used in SQL DML statements, just like Oracle's own built-in aggregates. Once such functions are registered with the server, Oracle simply invokes the aggregation routines that you supplied instead of the native ones. User-defined aggregates can be implemented using `ODCIAggregate` interface routines.

You can create a user-defined aggregate function by implementing a set of routines as methods within an object type, so the implementation can be in any Oracle-supported language for type methods, such as PL/SQL, C/C++, or Java. When the object type is defined and the routines are implemented in the type body, you use the CREATE FUNCTION statement to create the aggregate function.

Each of the four `ODCIAggregate` routines required to define a user-defined aggregate function codifies one of the internal operations that any aggregate function performs, namely:

- Initialize - Initializes the computation;
- Iterate – processes each successive input value;
- Merge – combines two aggregation contexts and returns a single aggregation context;
- Terminate – computes the result.

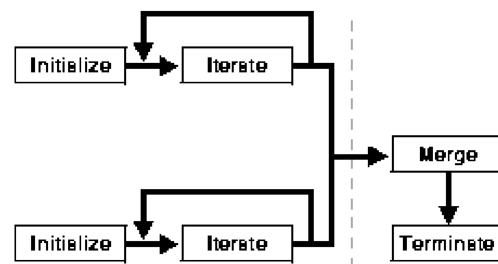


Fig. 9. ODCIAggregate routines [9]

Keith Laker's proposed user-defined function is developed in the following manner:

An initial object is created with the purpose of storing the results from the iterate stage. In this example, the object is created as a table of *varchar2(25),* but the size can be up to 4000 bytes, even *clob*, depending on the way the final aggregate function is being used. For example, if we plan on concatenating first names and last names, this object should have the size of the two columns combined.

```
CREATE OR REPLACE TYPE string_varray AS TABLE OF VARCHAR2(25);

CREATE OR REPLACE TYPE t_string_agg AS OBJECT
(
```

```
 a_string_data string_varray,
 STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT t_string_agg) RETURN
NUMBER,
 MEMBER FUNCTION ODCIAggregateIterate(self IN OUT t_string_agg, value IN
VARCHAR2 ) RETURN NUMBER,
 MEMBER FUNCTION ODCIAggregateTerminate(self IN t_string_agg, returnValue
OUT VARCHAR2, flags IN NUMBER) RETURN NUMBER,
 MEMBER FUNCTION ODCIAggregateMerge(self IN OUT t_string_agg, ctx2 IN
t_string_agg) RETURN NUMBER
);

CREATE OR REPLACE TYPE BODY t_string_agg IS
 STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT t_string_agg) RETURN
NUMBER IS
 BEGIN
    sctx := t_string_agg(string_varray() );
    RETURN ODCIConst.Success;
 END;
MEMBER FUNCTION ODCIAggregateIterate(self IN OUT t_string_agg, value IN
VARCHAR2) RETURN NUMBER IS
 BEGIN
    a_string_data.extend;
    a_string_data(a_string_data.count) := value;
    RETURN ODCIConst.Success;
 END;
MEMBER FUNCTION ODCIAggregateTerminate(self IN t_string_agg, returnValue
OUT VARCHAR2, flags IN NUMBER) RETURN NUMBER IS
   l_data varchar2(32000);
   ctx_len NUMBER;
   string_max NUMBER;
BEGIN
   ctx_len := 0;
   string_max := 4000;
   FOR x IN (SELECT DISTINCT column_value FROM TABLE(a_string_data) order
by 1)
    LOOP
      IF LENGTH(l_data || ',' || x.column_value) <= string_max THEN
         l_data := l_data || ',' || x.column_value;
      ELSE
         ctx_len := ctx_len + 1;
      END IF;
   END LOOP;
   IF ctx_len > 1 THEN
        l_data := l_data || '...(' || ctx_len||')';
    END IF;
    returnValue := LTRIM(l_data, ',');
    RETURN ODCIConst.Success;
 END;
MEMBER FUNCTION ODCIAggregateMerge(self IN OUT t_string_agg, ctx2 IN
t_string_agg) RETURN NUMBER IS
 BEGIN
    FOR i IN 1 .. ctx2.a_string_data.count
     LOOP
        a_string_data.EXTEND;
        a_string_data(a_string_data.COUNT) := ctx2.a_string_data(i);
     END LOOP;
    RETURN ODCIConst.Success;
 END;
END;
```

The last step is creating a function – the actual *ListAgg* alternative, which receives a string as input and calls the string-processing object described above.

```
CREATE OR REPLACE FUNCTION string_agg (p_input VARCHAR2)
RETURN VARCHAR2
PARALLEL_ENABLE AGGREGATE USING t_string_agg;
```

We should observe that in `ODCIAggregateTerminate` routine, `string_max` is set to 4000. This basically reproduces the *ON OVERFLOW TRUNCATE* functionality of *ListAgg*'s 12c version.
The advantage of such an approach is that the function can directly answer the problem we face with a custom solution. The initial object can be as large or as small as we need it to be. The overflow truncate can occur at any given threshold. This versatile approach can be further used in any given SQL Statement, no matter its complexity.
One disadvantage could be that the separator is embedded within the code. The developer cannot change it as easily as with *ListAgg*.
Let us observe the user-defined aggregate in action (in the below example, `string_max` is set to 100):

```
select e.department_id,
       string_agg(e.first_name|| ' ' || e.last_name) as full_name
from employees e group by e.department_id;
```

	DEPARTMENT_ID	FULL_NAME
1	10	Jennifer Whalen
2	20	Michael Hartstein,Pat Fay
3	30	Alexander Khoo,Den Raphaely,Guy Himuro,Karen Colmenares,Shelli Baida,Sigal Tobias
4	40	Susan Mavris
5	50	Adam Fripp,Alana Walsh,Alexis Bull,Anthony Cabrio,Britney Everett,Curtis Davies,Donald OConnell...(38)
6	60	Alexander Hunold,Bruce Ernst,David Austin,Diana Lorentz,Valli Pataballa
7	70	Hermann Baer
8	80	Alberto Errazuriz,Allan McEwen,Alyssa Hutton,Amit Banda,Charles Johnson,Christopher Olsen,David Lee...(27)
9	90	Lex De Haan,Neena Kochhar,Steven King

Fig. 10. *ListAgg* functionality replicated with a user-defined function

4. Conclusions

ListAgg currently remains a powerful string aggregation alternative. Throughout their career, developers usually work with multiple versions of databases, which is why a review on *ListAgg* evolution, limitations, and string processing alternatives may prove useful.

References

[1] https://support.oracle.com/knowledge/Oracle%20Cloud/2068368_1.html
[2] https://docs.oracle.com/
[3] http://www.dba-oracle.com/t_oracle_listagg_function.htm
[4] https://oracle-base.com/articles/misc/string-aggregation-techniques#wm_concat
[5] https://blog.dbi-services.com/oracle-12cr2-sql-new-feature-listagg-overflow/
[6] https://rogertroller.com/2020/01/07/oracle-19c-listagg-enhancement/
[7] https://blogs.oracle.com/datawarehousing/exploring-the-interfaces-for-user-defined-aggregates
[8] https://asktom.oracle.com/pls/apex/f?p=100:11:0::::p11_question_id:15637744429336
[9] https://docs.oracle.com/cd/B10501_01/appdev.920/a96595/dci11agg.htm#1004615

Cristian DUMITRESCU graduated from the Faculty of Cybernetics, Statistics and Economic Informatics of the Bucharest University of Economic Studies in 2010. Cristian works as a Data Engineer for IBM Romania and has 7 years of experience in working with Oracle Databases. His area of expertise includes Relational Databases, SQL, PL/SQL, OLTP, OLAP, and Reporting.