Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 1. pp. 311–324.

Techniques for pagination in SQL

András Gábor

Department of Information Technology, Faculty of Informatics
University of Debrecen
e-mail: gabora@inf.unideb.hu

Abstract

Web applications are replacing the standard client-server architecture. Accessing data in the new architecture gave rise to new techniques which answer the questions of traditional information retrieval in a new manner. Such a very typical question is presenting an ordered list through pages in a web application: *pagination*.

A special case of pagination is paging through an ordered list of items belonging to a certain category. Examples of this are browsing the entries belonging to a topic in a forum ordered descending by their entry dates or presenting the goods in a web-shop belonging to a certain type of goods ordered alphabetically.

A straightforward solution to this in a traditional application is to keep open a cursor to the ordered set and navigate back and forth as needed. However in client/server architecture this is not a viable solution because the client usually does not inform the server about the end of its interest in these rows. Thus for each page of rows a separate SQL query is issued.

In this paper we define the problem statement of pagination and show various SQL techniques in the Oracle ORDBMS used by application developers nowadays. We point out the drawbacks and performance issues these answers have and show alternate techniques which are closer to an optimal solution. We can than compare the efficiency, the structural complexity, the ease of use, readability and manageability of these solutions.

Keywords: : pagination, paging, SQL, optimization, Oracle

1. Definitions

- Let T be a relational table with an ordered set of columns S. A set of rows – R in the table belong to the same category C if and only if the values in all columns in S have the same values in all rows in R. This is a classification of the rows in T. We will call this the categorization of T by S.

- Let T be a relation table with an ordered set of columns O. The rows of T can be ordered by O in direction D, where D is either ascending or descending. Therefore any subset of the rows in T can also be ordered by O in direction D.

Common tasks related to these two properties:

Task1 *Counting:* Determining the cardinality of rows in a certain category C in table T.

Task2 *Ordered Category Retrieval:* Retrieving the rows of a certain category C in table T ordered by O in direction D.

Task3 *Category Pagination:* Retrieving a range of rows from the rows of category C ordered by O in direction D from table T.

Generally speaking *pagination* is the task of retrieving a page of rows from an ordered set of rows determined by a query. The ordered set has no restrictions. The source can be a filtered table, a view or even a complex subquery. An optional part of the task is to number the rows.

From the above follows that *Task 3* is a special case of pagination, for simplicity we will use the name *category pagination*. Here the restriction is that the rows to be paged are from only one table T and the filtering is specified by the categorization C only.

The parameters of category pagination are:

- table T
- categorization C
- ordering O
- direction D
- range R

The range can be specified in multiple equivalent ways:

- first row number, last row number
- first row number, page size (length of range)
  $last\ row = first\ row + page\ size - 1$
- page number, page size
  $first\ row = (page\ number - 1) * page\ size + 1$
  $last\ row = page\ number * page\ size$
- first page number, last page number, page size
  $first\ row = (first\ page\ number - 1) * page\ size + 1$
  $last\ row = last\ page\ number * page\ size$

2. Real world examples

The first example is a web-forum, where the rows are stored in a forum table. The category is the topic. Entries can be ordered by creation date in descending order.

Category Pagination occurs as soon as the first page of the forum topic is presented. Then the client may page to the next page using an arrow link or a button. Most forums also have quick direct links that facilitate jumps to a specific page or to the last page.

To determine the total number of pages for any topic *Counting* is performed beforehand. Counting is also needed if the website needs to present the total number of entries in a topic. Depending on the ratio of queries to DML activity this can be optimized by having a redundant counter in the topic table or caching this value on the application side.

Other examples could be web-shops presenting lists of pre-categorized goods to the users.

Personally I have used the techniques discussed in this paper in a RDBMS based web messaging system to optimize performance. In my case the table stored the messages; categorizing column was the sender's or receiver's id. The messages were listed on pages ordered by their date descending (outbox / inbox).

3. Example data

To help formulate SQL queries and help measure the efficiency we create a sample table, populate it and create an appropriate index on it.

```
CREATE TABLE test_table (
   id              NUMBER
 , category_col    VARCHAR2(20)     NOT NULL
 , order_col       NUMBER           NOT NULL
 , fixed_data      CHAR(1000)       NOT NULL
 , variable_data   VARCHAR2(2000)   NOT NULL
 , last_col_data   CHAR(1)          NOT NULL
)
```

The *data* columns enlarge the row sizes similar to real world applications.
The sample data loaded has the following characteristics:

- 100,000 rows
- 20 categories
- Average 5,000 rows in each category with little variance
- Ordering values are unique
- Table rows are in 8KB blocks

- ≈ 2530 bytes average row length (column `variable_data` has random data of length 1000-2000 chars)
- ≈ 46000 blocks used
- It is very improbable for any 2 neighboring rows to reside in the same block (2 rows are neighbors: they are in the same category and follow each other in the ordering). Oracle terminology: the clustering factor is close to the number of rows.

We also create an index that allows the Oracle ORDBMS to range scan a category ordered in either direction.

```
CREATE INDEX idx_tt_cat_ord ON test_table(category_col, order_col) COMPRESS 1
```

In this Oracle B*-tree index the index entries contain the values of these 2 columns and the ROWID of the original row. The characteristics of the index for the sample data are:

- index is stored 8K blocks
- ≈ 18 bytes average index row length
- ≈ 390 blocks used (less than 1% of the number of blocks of the table), ≈ 250 leaf blocks
- ≈ 400 index entries / leaf block
- index height is 3 (blevel=2)

Since the tree is 3 levels high, it takes 3 block reads to access the first entry in the index based on some value. In a range scan the following rows will be likely in the same block still (≈ 400 index entries in a leaf block). So to access all the index entries of one category we need $3 + 5000/400 = 15.5$ blocks on the average.

4. SQL techniques in Oracle ORDBMS

In the following we show various SQL statements for *Category Pagination* and discuss their properties. The parameters are bind variables denoted by a colon (:) and named `P_`<*name*>.

We assume that it is not viable to store all columns in a single index, that is `INDEX(category_col, order_col,` <*rest_of_the_columns*>`)` nor in a similar index organized table (IOT). Also we assume that there is no other appropriately usable index on the test table.

For the efficiency we will explain the most probable execution plan used by the query optimizer. We measure the number of block reads (logical I/O) as the main performance metric and emphasize other consumed resources only when relevant.

We use at least 3 test cases. In all cases the page size is 10 rows, but the page numbers differ. We will query the 1st, 5th and 100th pages. We query the rows in ascending order for simplicity. The total number of the rows in the tested category is 5021.

Measurements can be carried out with standard Oracle tools such as the autotrace facility or querying the V$MYSTAT view. More fine-grained statistics can be collected using the more advanced autotrace facility in Oracle's SQL Developer [6] or analyzing data populated in the V$SQL_PLAN_STATISTICS_ALL view.

5. Standard techniques

The following techniques are used and are known by most of the developers.

1. Full sort / no sort, fetch until last row

```
SELECT ROWNUM rn, t.*
 FROM (SELECT t.*
        FROM test_table t
        WHERE category_col = :P_CATEGORY
        ORDER BY order_col
       ) t
```

•*Rationale:* Using the above query the application counts the rows and fetches until the last row on the page. The rows retrieved before the first row on the page are discarded on the application side, whereas the rows after the last row are never really retrieved.

This solution also solves *Task 2 Ordered Category Retrieval*.

•*SQL Execution:* The rows are filtered using the index and are returned in the correct order because of the index access, no sorting is done. After that all the rows that are really fetched get looked up using their ROWIDs retrieved from the index. For the rows after the last row fetched neither their index entry nor their row entry from the table block is read.

Let us consider another case when our index contains only the `category_col` (and not `order_col`).

This index still can be used to filter the rows but the RDBMS now has to sort the rows as well. For this it has to read all the rows in the category, regardless of the fact that not all of them are to be fetched by the application. We consider this case because in many real world systems only the filtering columns are indexed, which is a mistake.

•*Efficiency:* Having an index to perform both filtering and ordering makes this query much cheaper than with a filter-only index.

With the filter-only index a constant number of block reads is done regardless of the page number. With the concatenated index the number of block reads is linear with the page number (disregarding the initial 3 block reads needed to "jump" into the index).

Both ways we waste time and network resources while fetching the rows before the desired page and also the block reads needed to access these rows.

Page#	1		5		100	
Index (Category only/Concatenated)	Cat Only	Concat	Cat Only	Concat	Cat Only	Concat
# of block reads	4890	22	4890	102	4890	2002

2. Full sort / numbering, fetch page rows only

```
SELECT t.*
  FROM (SELECT ROWNUM AS rn, t.*
        FROM (SELECT t.*
                FROM test_table t
                WHERE category_col = :P_CATEGORY
                ORDER BY order_col
              ) t
       ) t
  WHERE rn BETWEEN :P_FIRST_ROW AND :P_LAST_ROW
```

•*Rationale:* We refine the previous SQL and do a filtering on the database side so that only the relevant rows are returned by the query.
•*SQL Execution:* The rows are filtered using the index and are returned either in order using the concatenated index or by sorting. Then all rows in the category are numbered. Finally the rows having an inappropriate row number are filtered out (thrown away).
•*Efficiency:* For all the rows in the category all the index entries and the table row entries are read for the numbering. Having a concatenated index saves us the time and resources needed for sorting, but still all rows are accessed for the numbering. Notice that the processes of numbering and filtering are related but the RDBMS fails to recognize and exploit this fact.

From now on we only investigate cases in the presence of the concatenated index only.

Page#	All cases: 1, 5, 100
# of block reads	5038 (= 17 index + 5021 table)

3. Top-N query, extended extremal values search

```
SELECT t.*
  FROM (SELECT ROWNUM AS rn, t.*
        FROM (SELECT t.*
            FROM test_table t
            WHERE category_col = :P_CATEGORY
            ORDER BY order_col
             ) t
        WHERE ROWNUM <= :P_LAST_ROW
         ) t
WHERE rn >= :P_FIRST_ROW
```

•*Rationale:* We refactor the previous solution and specify the last row filtering before the numbering column is renamed. In Oracle this is called a Top-N query.

It has the formal criterion that the form `ROWNUM <=` *<expression>* must be used and this must be applied on the ordered subquery (inline view) without further nesting. We cannot do the same with our other condition as `ROWNUM >= :`x where $x > 1$ would filter out all rows since there can be no x^{th} row without having a 1^{st} row before.

•*SQL Execution:* Oracle recognizes the Top-N query form and will do a sorting which is a modified extremal values search i.e. only the first N rows are stored in memory at any time during the sort using usually much less memory and CPU than a full sort would need. In our case the sorting is omitted totally because of the proper index. The Top-N condition is evaluated via fetching only the first N index entries. The internal step is called "COUNT (STOPKEY)" meaning actually that rows are not to be accessed after a certain number of rows get retrieved.

•*Efficiency:* This solution filters out the unnecessary leading rows on the database side saving network resources and fetch time. The number of block reads is linear with the page number.

Page#	1	5	100
# of block reads	13	53	1006

4. Numbering with analytical functions (full sort), fetch relevant rows only

```
SELECT t.*
  FROM (SELECT ROW_NUMBER() OVER (ORDER BY order_col) AS rn, t.*
           FROM test_table t
           WHERE category_col = :P_CATEGORY
       ) t
WHERE rn BETWEEN :P_FIRST_ROW AND :P_LAST_ROW
```

•*Rationale:* Numbering can be done using the analytical function `ROW_NUMBER()`.

•*SQL Execution:* This is another form of the Top-N query that Oracle can recognize. The syntax is much more convenient and more expressive.

•*Efficiency:* The internals are only slightly different. Here a "WINDOW (NOSORT STOPKEY)" operation is used instead of the "COUNT (STOPKEY)". The option "NOSORT" refers to the fact that rows are retrieved from the index in the correct order.

Page#	1	5	100
# of block reads	15	55	1008

In practice developers often include the `PARTITION` phrase in the numbering analytical clause unnecessarily:

a) `ROW_NUMBER() OVER (PARTITON BY category_col ORDER BY order_col)`

b) `ROW_NUMBER() OVER (PARTITON BY 'x' ORDER BY order_col)`

These are logically equivalent with the original version however Oracle does not any more recognize the Top-N form. Instead it retrieves all the rows and performs the numbering on all of the rows and only after that comes the filtering, the same way as in Solution 2 resulting exactly the same high number of block reads.

Page#	All 3 cases (page#: 1, 5, 100) when `PARTITION BY` also specified
# of block reads	5038(= 17 index + 5021 table)

6. Optimal techniques using pure SQL

The above techniques are used and promoted among developers. However these techniques are far from optimal. The best solutions so far are the Top-N queries. Internally these still access all the rows until the last row, including the rows before the selected page.

Our queries are not really Top-N queries; they could rather be called "Top-N - Last-M" queries, where $M \leqslant N$.

In this section we show techniques that eliminate the access of the rows preceding the desired page.

5. ROWID index lookup with self join, real "Top-N - Last-M" query

```
SELECT i.rn, t.*
  FROM (SELECT i.*
          FROM (SELECT i.*, ROWNUM AS rn
                  FROM (SELECT ROWID AS a_rowid
                          FROM test_table t
                          WHERE category_col = :P_CATEGORY
                          ORDER BY order_col
                        ) i
                  WHERE ROWNUM <= :P_LAST_ROW
        ) i
        WHERE rn >= :P_FIRST_ROW
    ) i
    , test_table t
    WHERE i.a_rowid = t.ROWID
    ORDER BY rn
```

•*Rationale and SQL Execution:* In the innermost subquery we only use data from the table that is also available in the index: the 2 columns and the ROWID. Hence Oracle needs to access the index entries only, reading a few blocks for the ordering (done by the index actually), the numbering and the filtering. In the numbering part we also leverage the Top-N form. At the outermost level the inline view i contains the row numbers and ROWIDs of the rows on the page only i.e. *page size* rows only. In our case this is 10 ROWIDs. At this point the join is performed using a loop where based on the 10 ROWIDs a table lookup is performed.

•*Efficiency:* We look up the index entries with a Top-N query. During this only a minimum number of blocks are accessed since no table block is accessed within the inline view i. Then based on the resulting at most *page size* ROWIDs the table blocks are accessed, resulting in exactly *page size* block reads (provided that the each row is stored in at most one block). The trick here is to perform both the Top-N and the Last-M filter on the index before actually accessing the rows. This

way when accessing a far page the number of block reads will increase only with a few extra block reads.

The number of block reads is linear with the sum of the page size and a small fraction of the page number. Even when accessing the 400$^{\text{th}}$ page we only access 23 blocks!

Page#	1	5	100	400
# of block reads	13	13	16	23 (!)

Note: This solution works less efficiently with index organized tables (IOTs) because the ROWIDs stored in the secondary indexes of IOTs contain a guess ROWID and the PRIMARY KEY value. Using the above SQL we lose the quess ROWIDs before the join and therefore a PRIMARY KEY based lookup will occur turning each ROWID based lookup into an index unique scan. However Solution 6 and 7 does work for normal tables and IOTs in the same efficient way.

6. Index based first row value lookup with one page fetch forward

```
SELECT ROWNUM + :P_FIRST_ROW - 1 AS rn, t.*
  FROM (SELECT t.*
        FROM test_table t
        WHERE category_col = :P_CATEGORY
        AND order_col >= (SELECT order_col
                          FROM (SELECT order_col, ROWNUM rn
                              FROM (SELECT order_col
                                  FROM test_table t
                                  WHERE category_col = :P_CATEGORY
                                  ORDER BY order_col
                                    )
                              WHERE ROWNUM <= :P_FIRST_ROW
                                )
                          WHERE rn = :P_FIRST_ROW
                        ) - first item on page
           ORDER BY order_col
         ) t
WHERE ROWNUM <= :P_PAGE_SIZE
```

•*Rationale and SQL Execution:* The idea here is to look up the value of the *order_col* column in the first resulting row and then based on this value scan a page size of rows with a normal index range scan. This is achieved in 2 steps. Take a look at the condition `order_col >= (`*<subquery>*`)`. Here the subquery determines the `order_col` value in the first row of the result page using the last value from a Top-N query where the data can be fetched from the index only. So here again the table rows are not accessed at all in the innermost subquery, because all the data i.e. the 2 columns are retrieved from index. After retrieving this starting value a normal lookup can be done, just as if this value was a literal or a variable. So using this value Oracle can jump into the index to the starting position and do an index range scan followed by a table access by index ROWID. From that point the first *page size* rows can be retrieved efficiently using the Top-N form again.

•*Efficiency:* The price of the query is the sum of the price of the subquery and the range scan of one page of rows. The subquery uses the index only; therefore the number of block reads is linear with a small fraction of the page number. The number of block reads in the rest of the query is linear with the page size. No table rows are accessed unnecessarily. The only difference in efficiency compared to Solution 5 is that the index must be accessed using a lookup value twice. In this concrete case this adds constant 3 block-reads to the results.

Page#	1	5	100	400
# of block reads	16	16	19	26

Note: Solution 6 has one great advantage and one small disadvantage over Solution 5.

Advantage: In Solution 6 the ROWIDs retrieved from the index are only used internally; when applying this solution to IOTs the guess ROWIDs are utilized efficiently. This solution has been applied by me in the largest social networking web site in Hungary.

Disadvantage: Solution 6 cannot be used as is when the ordering column set (`order_col`) is not unique within all categories. Consider the case when the same value is in the last row on a page and in the first row of the next page. A modified usually less efficient version of the query could be used by suffixing the `order_col` with the ROWID or another unique key stored in the index.

Summarizing Solution 5 and 6

Solution 5 and 6 both use very few block reads. The only inefficiency in them is the access of the index entries of the rows before the desired page. This could be eliminated with an index with columns:
`category_col`, `row_number_in_category_ordered_by_order_col`.

However maintaining such a row number value is not automatic. Every DML modifying the ordering column may trigger the recalculation of the row number of many other rows in the same category. Therefore this is only acceptable in environments where individual row modifications of the `order_col` column including insertions and deletions are extremely rare.

7. Better techniques using ORDBMS extensions

The SQL language is said to be declarative, however we just abused its syntax to drive the optimizer's decisions. The structural complexity of the solutions is far from being straightforward to any developer and is contradictory to the very essence of the SQL language. Therefore we must look for and find better, more expressive ways.

The basic idea is to abstract away the complex functionality and encapsulate it within a subprogram-like construct that can be named and parameterized intu-

itively.

7. Using a custom domain index

```
SELECT ROWNUM + :P_FIRST_ROW - 1 AS rn, t.*
  FROM test_table t
  WHERE category_page('category_col, order_col', :P_CATEGORY
                , :P_DIRECTION, :P_PAGE_SIZE, :P_PAGE_NUMBER) > 0
```

•*Rationale:* The Oracle RDBMS has a facility called extensible indexing. We can create an object type implementing an interface ODCIIndex[1] in which we implement the tasks associated with our custom indexing method. We have to define methods to implement the maintenance tasks i.e. creation, rebuilding and dropping tasks. But most importantly we are to provide an algorithm to retrieve the ROWIDs of all the rows that fulfill the search criterion associated with our indexing scheme.

Besides the object type we have to create at least one operator, a special function that can be used in SQL statements later. We named our operator *category_page*. Then an index type must be created to associate the operator(s) and the indexing object type.

Now indexes of the new index type can be created simply on any table:

```
CREATE INDEX idx_cat_ord_catpage
  ON test_table('category_col, order_col')
  INDEXTYPE IS category_page_idxtype
```

The actual index structures are created behind the scene via calling an ODCIIndex-interface method. In our case we simply create the concatenated index in this method.

•*SQL Execution:* The optimizer recognizes the operator and looks up the object type associated with it. Then the ROWIDs are fetched using 3 methods. The method *ODCIIndexStart* is called first in which we can open a helper cursor or context. After this the method *ODCIIndexFetch* is called which returns an array of the ROWIDs. The function is called repeatedly when not all the ROWIDs fit into one array. At the end the method *ODCIIndexClose* is invoked to allow for proper context cleanup.

We solve the task of category pagination by executing the index-lookup subquery from Solution 5 behind the scene. Based on this query we internally retrieve the ROWIDs from the concatenated index.

•*Efficiency:* In terms of block reads this solution is equivalent with Solution 5 additionally it works efficiently with IOTs as well. Instead of having a join the RDBMS executes our custom code which is naturally slightly slower using slightly more resources.

[1]ODCI stands for Oracle Data Cartridge Interface [5].

•*Important:* Once the domain index code is installed in a concrete Oracle system it can be used very easily and straightforwardly multiple times with the same and/or different tables. The SQL statements are simple, clean and expressive again. Oracle has long been using domain indexes to implement full text indexing and multimedia indexing.

Example domain index implementation is available from my website [7].

8. Table functions

We can hide the complex code of the pagination logic behind table functions also:

a) Using a table function that returns row numbers and ROWIDs from any table:

```
SELECT i.rn, t.*
 FROM TABLE( <table function>(<table name>, <pagination parameters>) ) i
    , test_table t
WHERE i.a_rowid = t.ROWID
ORDER BY rn
```

b) Using a table function that return entire rows of a specific table:

```
SELECT ROWNUM + :P_FIRST_ROW - 1 AS rn, t.*
  FROM TABLE(<table function for table T>(<pagination parameters>) ) t
ORDER BY order_col
```

Example implementations are available from my website [7].

Both solutions require that the concatenated index is created separately.

In Solution 8.a) the function returns (`NUMBER, ROWID`) pairs regardless of the structure of the paged table. However the implementation is not so elegant because we still have to do a join.

In Solution 8.b) the function's return type must have the same structure the table has. Hence for every different table we must create a proper object type and a collection type and a corresponding function as well. This is quite poor code reusability.

•*Efficiency:* Internally both functions use Solution 5 or 6; therefore the number of block reads is equivalent. The extra task here is the cost of running the custom codes and the cost of creating object instances during runtime.

9. Cursor functions

Besides table functions we can return multiple rows using functions returning cursors. The disadvantage of cursor functions is that they cannot be used as subqueries (e.g. in the FROM clause of a query) because the exact structure of the rows returned is not known to the query optimizer at compile time.

However this is not a strong restriction usually. The cursor returned by the function can be used on the client side the exact same way as the result of any other query. Also no intermediary object instances are needed during runtime.

Example client side usage in Java:

```
CallableStatement call = connection.prepareCall(
   " ? = <cursor function>(<table name>,
<pagination parameter placeholders>");
call.registerOutParameter(1, OracleTypes.CURSOR);
call.set<Type>(1, <pagination parameter1>);
call.set<Type>(2, <pagination parameter2>);
...
call.execute();
ResultSet rs = (ResultSet) call.getObject(1);
...
```

Example implementation of the cursor function is available from my website [7].

8. Future work

There are several directions we need to continue our studies:

1. We have to perfect the domain index and the function-based implementations. In the case of the domain index this means testing and usability enhancements like supporting other types besides NUMBER, supporting multiple columns for categorizing and ordering, etc. For the function-based solutions we are to create a framework to generate new types and table functions for any table.

2. Currently the queries are restricted to the categorization criterion only. There are sometimes additional filtering criteria applied before paging. This could be addressed within the domain index/function-based implementations making a step further to solving the problem of the general pagination efficiently.

3. We have been experimenting with hierarchical multilevel index structures in order to derive exact row numbers. This could be beneficial in circumstances where very long lists are paged through thousands of pages by saving the index entry reads preceding the desired page. Such indexes could also be used to support *Task 1 Counting*.

4. We have to investigate the available techniques in other RDBMSs helping the industry to decide on a product when pagination is a main task and also helping the developers of these systems to enhance their products to better support pagination in the future.

References

[1] Kyte, T., On Top-n and Pagination Queries, *Oracle Magazine* (Jan./Feb. 2007) 63–66. http://www.oracle.com/technology/oramag/oracle/07-jan/o17asktom.html

[2] Gábor, A., Juhász, I., PL/SQL-programozás. Alkalmazásfejlesztés ORACLE10g-ben, *Panem,* Budapest, (2007). (Title in English: PL/SQL-programming. Application Development in ORACLE 10g.)

[3] Gábor, A., Gunda, L., Juhász, I., Kollár, L., Mohai, G., Vágner, A., Az Oracle és a WEB. Haladó Oracle9i ismeretek, *Panem,* Budapest, (2003). (Title in English: Oracle and the WEB. Advanced Oracle 9i.)

[4] Gábor, A., Database systems benchmarking, *6th ICAI,* Eger, Vol. 2, (2004), 73–81.

[5] Oracle®Database – Data Cartridge Developer's Guide, 10g Release 2 (10.2)

[6] Oracle SQL Developer, http://www.oracle.com/technology/products/database/sql_developer/ Implementation (code, scripts, exec-plans): http://www.inf.unideb.hu/~gabora/pagination/