Computing Frequent Itemsets Inside Oracle 10G

Wei Li

Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A

weili@oracle.com

Ari Mozes

Oracle Corporation
10 Van de Graaff
Burlington, MA 01803
U.S.A

ari.mozes@oracle.com

Abstract[1]

Frequent itemset counting is the first step for most association rule algorithms and some classification algorithms. It is the process of counting the number of occurrences of a set of items that happen across many transactions. The goal is to find those items which occur together most often. Expressing this functionality in RDBMS engines is difficult for two reasons. First, it leads to extremely inefficient execution when using existing RDBMS operations since they are not designed to handle this type of workload. Second, it is difficult to express the special output type of itemsets.

In Oracle 10G, we introduce a new SQL table function which encapsulates the work of frequent itemset counting. It accepts the input dataset along with some user-configurable information, and it directly produces the frequent itemset results. We present examples of typical computations with frequent itemset counting inside Oracle 10G. We also describe how Oracle dynamically adapts during frequent itemset execution as a result of changes in the nature of the data as well as changes in the available system resources.

1. Introduction

Frequent itemset counting operation is a common data mining operations and it is used for several important mining algorithms, such as association rules and large bayes classification.

Frequent itemset counting has been directly used in business-intelligence for many years. The most common application has been in the retail industry in determining which products are most commonly purchased together. For instance, people who buy beer are likely to buy pretzels. This is known as Market Basket Analysis.

In the Market Basket Analysis model, a transaction contains a limited number of items that occur together. Let $\boldsymbol{A}$ be a set of m items: $\boldsymbol{A} = \{\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_m\}$and $\boldsymbol{T}$ be a set of n transactions: $\boldsymbol{T} = \{\boldsymbol{T}_1, \boldsymbol{T}_2, \ldots, \boldsymbol{T}_n\}$. Frequent itemsets are subset of items in $\boldsymbol{A}$ which occur together in $\boldsymbol{T}$ above a predefined support threshold. If the data is stored organized by transaction, where each transaction has a number of items, we define this as a horizontal layout. Otherwise, if the data is stored organized by item, where each item has a number of supporting transactions, we define this as a vertical layout.

R. Agrawal, T. Imielinski, A. Swamy introduced association rules and the Apriori Algorithm in [1]. The Apriori Algorithm, which is based on the horizontal layout, is an iterative algorithm that has two steps in each iteration:

- generating candidate itemsets using Apriori-rule
- reading all the transactions, and for each combination of items within a transaction, incrementing a counter for that combination of items

A. Savasere, E. Omiecinski, S. Navathe introduced a different algorithm that is based on the vertical layout in [6]:

- a list of transactions for each item is created
- an itemset can be counted by simply intersecting all the involved items' transaction lists

(1) *Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

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

Most of the research literature is based on these two categories of algorithms.

Given the nature of the operation, producing frequent itemsets using existing SQL functionality is not trivial. In [4], S. Sarawagi, S. Thomas, R. Agrawal discussed various approaches to integrate the operation into RDBMS systems. We decided to push the functionality into the RDBMS engine for the following reasons:

- We want to provide end-users a simple and flexible interface yielding optimal performance.
- The SQL implementation stays closer to the dataset and avoids huge raw data transfers to and from the RDBMS system.
- We can closely interact with other built-in database functionality such as dynamic memory management and parallel execution.

Because we support frequent itemset counting as an internal database operation, we face several problems that are not considered in the research literature, including:

- output format: an itemset is a set of items, which does not have a corresponding native datatype in a pure RDBMS system
- resource usage: we can not use an arbitrary amount of resources (e.g., memory); we must consider the situation that the operation is running inside complex queries and, at the same time, there may be many other queries running inside the RDBMS system vying for system resources

The rest of this paper is organized as follows. In section 2, we describe our SQL interface. In section 3, we present our general execution ideas, including algorithms and adaptive execution. We give the demo content in section 4.

2. SQL interface - Table Functions

We support the frequent itemset counting operation in the form of a table function in Oracle 10g. We will describe the interface, using a simple example from [7].

Suppose that you are a product manager in Oracle. One of your job responsibilities is to write technical white papers. You have access to simple web reports that show you how many people have downloaded each piece of collateral, so you know which papers are popular.

Assume that you have a *web_log* table with three columns:

- *session_id*: downloading session ID
- c*ommand*: downloaded white papers
- *time_stamp*: session start time

You can use the following simple query to find the six most popular white papers downloaded from the website during a fixed time period:

```
select command,
       rank() over (order by support desc) rnk
from (select command, count(session_id) support
      from web_log
      where time_stamp between '01-APR-2002'
                      and '01-JUN-2002'
      group by command)
where rnk <= 6;
```

And you get the following results:

```
White paper title                           #
--------------------------------------------
Table Compression in Oracle9i               1
Field Experiences with Large Data Warehouse2
Key Data Warehouse Features                 3
Materialized Views in Oracle9i              4
Parallel Execution in Oracle9i              5
Query Optimization in Oracle9i              6
```

But now you want to go one step further and find the most common pairs of collateral downloaded in a single session. You can use the frequent itemset table function in Oracle 10g to answer the question:

```
select itemset,
    rank() over (order by support desc) rnk
from
(select cast(itemset as fi_char) itemset,
       support
from
table(dbms_frequent_itemset.fi_transactional(
    cursor(select session_id, command
           from web_log
           where time_stamp between
             '01-APR-2002' and '01-JUN-2002'),
    (60/2600), 2, 2))
where rnk <= 3;
```

The above query uses the *dbms_frequent_itemset* package[2] to find the most frequently downloaded pairs of papers. We use a nested table type to represent the output itemsets. Here, the nested table type is *fi_char*, which is defined as:

```
create type fi_char as table of varchar2(50);
```

The following results yields interesting findings. Although the white paper "Table Compression in Oracle9i" is the most popular individual paper, only one of the top three pairs includes this paper. Neither of the papers in the second most popular pair show up as individual items above.

```
White paper titles                          #
--------------------------------------------
Table Compression in Oracle9i and           1
Field Experiences with Large Data Warehouse
Data Warehouse Performance Enhancements and2
Oracle9i Performance and Scalability in DSS
Materialized Views in Oracle9i and          3
Query Optimization in Oracle9i
```

(2) You can reference [8] for the detailed syntax and examples about *dbms_frequent_itemset* package.

3. General Execution

In section 1, we mentioned that there are two categories of frequent itemset counting algorithms: horizontal-layout and vertical-layout based. We implemented versions of both algorithms for the internal operation in Oracle 10g. We were surprised to find that there is no research describing how to execute those algorithms with a limited amount of memory and how changes in available memory can greatly influence the performance of the algorithms.

Horizontal layout algorithms: There are three reasons that this category remains promising:

- In almost all situations, users store their information in this layout. Therefore, horizontal based algorithm can avoid the costly transformation from horizontal layout to vertical layout.
- The algorithm is very efficient for sparse datasets, which is reinforced by the fact that the average number of items per transaction is less than 20.
- The memory requirement is based on the candidate itemsets, not the transaction database. This behavior is desirable in some situations. For example, if we have a huge transaction database and a small set of candidate itemsets, the algorithm can work efficiently with a limited amount of memory.

Vertical layout algorithms: It has been found that vertical layout algorithms([3] [5] [6]) work better than horizontal layout algorithms in many situations.

The general idea of the vertical layout algorithm is to compute lists of transactions for each item and then intersect these lists to compute the number of transactions that have those items in common. There are two possible representations for each item's list of transactions:

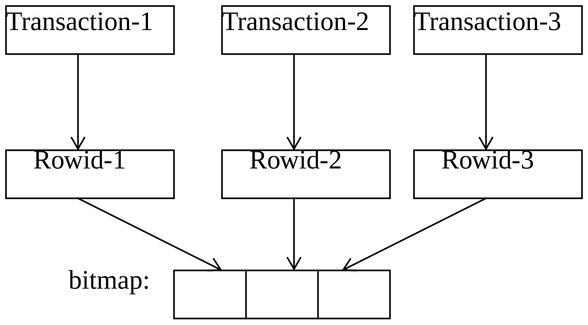
- linked list of all the transaction IDs
- bitmaps: each bit represents whether the corresponding transaction contains the item

We chose a bitmap representation for the following reasons:

- In our analysis, an intersection-based algorithm is most efficient when working on dense datasets. Bitmaps are a better representation for dense datasets.
- We can leverage proven Oracle bitmap index technology.

For frequent itemset computation, only the final count for a given itemset is of interest; there is no need to preserve information regarding which transactions contributed to a given itemset. Thus, to enable bitmap index representation, a simple mapping function is designed to map each transaction ID to a unique rowid, which results in each bit position being mapped to an actual transaction. After the mapping, we can utilize existing bitmap index counting and intersection techniques. Figure 1 shows the mapping process.

Figure 1 **transaction IDs -> bitmap Mapping**



The major drawback of the vertical layout approach for itemset counting comes from its large memory requirement. No matter whether the algorithm uses bitmaps or transaction lists, both of which can be compressed representations, the data is proportional to the original transaction table in size. Considering that the original transaction table may be gigantic, there is no guarantee that we can fit all of this data in memory.

Adaptive Execution: In our study, neither algorithm is clearly better than the other; the actual data, selection criteria, and available system resources will favor one over the other. A complication is that frequent itemsets are usually computed in phases where each successive phase increases the length of the itemset being processed; while one algorithm may be most efficient for one phase, the same algorithm may be quite inefficient for another phase. Due to the varying nature of the system resources and itemset combinations, our implementation adapts during execution, both within an algorithm and across algorithms.

To facilitate choosing the correct algorithm, during the first phase of execution we collect statistics that estimate the run time costs of both approaches. Then, for all of the remaining phases, we execute frequent itemset counting using the following steps to make it a self-tuning operator:

- Generate candidate itemsets using Apriori-rule. Statistics on candidates will also be collected.
- Using the Oracle query optimizer in conjunction with the collected statistics, estimate the IO and CPU costs of both algorithms. Comparing the two final costs, we proceed with the cheaper algorithm. At a very high level, the costs are as follows:

```
- Horizontal Cost = transactions scan cost +
                    counting tree setup cost +
                    recursive counting cost
- Vertical Cost = bitmap scan cost +
                  bitmap intersection cost +
                  bit counting cost +
                  final aggregation cost
```

Adaptation does not cease once an algorithm is chosen for a particular phase. A phase may take a long time to complete, and available memory in an RDBMS system will always be changing as users connect and disconnect from the database, firing off queries of varying complexity. There is a definite need to adapt within a phase as the utilization of the overall system changes.

To enable this type of intra-algorithm adaptive execution, two things were done in the Oracle implementation. First, the operator was written to take advantage of the ability to grow memory dynamically and to graciously handle the situation where memory usage needs to be reduced. Second, the operator cooperates fully with the Oracle memory manager regarding its memory usage.

As an example of adapting to memory constraints within the vertical layout approach, let us take the following scenario. We have 26 frequent items, item-A to item-Z, and a transaction database of 8 million transactions. Let us further assume that we have 10MB memory and can only fit 10 full items in memory. In general, we can not assume item-A and item-Z are in memory at the same time. One solution is to create a lightweight index on all of the items' bitmaps. Given the candidate itemset (A, B, Z), if only item-A and item-B are in memory, we can search in the index, find the position of item-Z, read the corresponding bitmap into memory (replacing another bitmap), and intersect all of the bitmaps to get the final counts. The cost of random I/O as used in this approach can be prohibitive, so this approach has clear performance issues.

We propose partitioning the bitmaps to address this memory constraint. First, bitmaps are partitioned according to available memory in such a way that the portion of each bitmap corresponding to a single set of transaction IDs fits in memory. Then, we intersect these partitions to produce partial itemset counts for each of the candidate itemsets. To produce complete aggregates, we will add another aggregation operator on top to sum across the partition subtotals. We must take care not to produce very small partitions since such an approach will degrade the good CPU performance of bitmap intersections and will be costly from an I/O perspective due to many small reads and writes.

When working with the Oracle memory manager, we define three types of executions for this partitioned bitmap intersection:

- *cache execution*: all bitmaps fit in memory
- *one-pass execution*: at least one partition of all bitmaps fits in memory
- *multi-pass execution*: even a single partition of bitmaps cannot completely fit in memory, in which case we propose to avoid partitioning and simple perform random access to retrieve a given bitmap from the index

Throughout execution, the frequent itemset counting operator will communicate with the Oracle memory manager [9], providing the memory requirements for its three possible mechanisms for execution, retrieving the memory bound, and deciding which execution method to use. The above example detailing the use of partitioned bitmaps is just one area in which the frequent itemset counting operator interacts with the memory manager to dynamically adjust to available resources.

4. Demo content

The demo will contain experiments for both a real world dataset and a synthetic dataset. We will show the benefit of integrating this functionality within the RDBMS: users can access and post-process results in the relational query environment. In addition, we will show how frequent itemset execution communicates with the Oracle memory manager so as to adapt to the current environment.

References

[1] [RTA93] "Mining association rules between sets of items in large databases", R. Agrawal, T. Imielinski, A. Swamy, ACM SIGMOD Conf. 1993

[2] [RJ96] "Parallel mining of association rules", R. Agrawal, J. Shafer, IEEE Trans. on Knowledge And Data Engineering, 1996

[3] [MJZ97] "New algorithms for fast discovery of association rules", M. J. Zaki, Int'l Conf. on Knowledge Discovery and Data Mining, 1997

[4] [SSR98] "Integrating association rule mining with relational database systems", Alternatives and Implications, S. Sarawagi, S. Thomas, R. Agrawal, ACM SIGMOD Conf. 1998

[5] [PJH00] "Turbo-charging vertical mining of large database", by P. Shenoy, J. Haritsa, S. Sudarshan, ACM SIGMOD Conf. 2000

[6] [AES95] "An efficient algorithm for mining association rules in large databases", A. Savasere, E. Omiecinski, S. Navathe, Proc. 21st Int'l Conf. Very Large Databases, San Francisco, 1995

[7] [FIO03] "Frequent itemsets in Oracle 10g", 2003, www.otn.oracle.com/products/bi/pdf/10gr1_twp_bi_dw_freqitemsets.pdf

[8] [O10G] Oracle 10G PL/SQL Packages and Types Reference

[9] [BM02] SQL Memory Management in Oracle9i, Proc. 28th Int'l Conf. Very Large Databases, Hong Kong, 2002