How to avoid a salted Banana

Lothar Flatz
Centris AG
Solothurn

Keywords
Performance tuning, Oracle, Execution Plan, Index Scan, throw away, efficiency

Introduction
We address a challenging issue in the context of sql statement tuning and optimal execution plans. There is one category of queries that is very difficult to tune. These are queries with independent and not very selective search criteria on two (or more) tables.
One prominent example is an address search when based on a person's name and a street address. These queries are characterized by a potentially high number of throw away rows. Throw away rows are retrieved, but do not make it into the final result.
We will see a patented technique describing how we can keep the effort to retrieve throw away rows at a reasonable level.

A Joke that highlights an issue
The title refers to a joke - told at the beginning - that quintessentially states that you should not make an extra effort for something you are going to throw away.
Two men sit together in a train. One of them takes out a banana, peels it, salts it, and throws it out of the window. Then he takes another banana and repeats the procedure.
After the third banana the other man cannot further hold his curiosity in bay and asks: “Why do you throw all these banana out of the window?” ”Well,” the other man replies, “do you like salted banana?”

Before we move on to more serious stuff, let us hold on for a moment of contemplation. Why is this joke funny?
Well, if you want to throw away your banana, why would you go through the process of peeling and salting it first? But is it possible that we do something similar in real life, for example in database performance tuning?

The concept of throw away in performance tuning
Throw away rows are rows that were retrieved by the database, but not used in any way to compute the result of a query.
Minimizing throw away rows is a common concept of performance tuning [1], paragraph 12.1.2.
The concept is intuitively plausible, but a row that is retrieved, but not used is certainly lost effort.

The most common use of the throw away concept is index optimization, as shown in following figure.

Operation	Actual Rows
SELECT STATEMENT	1
SORT ORDER BY	1
PARTITION RANGE ALL	1
TABLE ACCESS BY LOCAL INDEX ROWID	1
INDEX RANGE SCAN	175K

Fig. 1: sub optimal index access indicated by throw-away

Here you see the essential part of a sql monitor output. We see an index access retrieving 175k rows. When the table is accessed, only one row remains.

There must be an additional filter on the table that reduces the rows to that level. Certainly that filter is missing in the index. All we need to do look up this filter in the execution plan and add it to the index. (Note: of course we must also make sure that the additional field in the index will be commonly used and not just by one query.)

This way we would have the throw away in the index access as opposed to have it in the table access.

So why would it be better to have the throw away in the index? Let us have a look on the next figure.

Here we see the exact same execution plan as the one above, but this time focused on physical I/O.

Operation	IO Requests
SELECT STATEMENT	
SORT ORDER BY	
PARTITION RANGE ALL	
TABLE ACCESS BY LOCAL INDEX ROWID	8,337
INDEX RANGE SCAN	431

Fig. 2: I/O wait time compared: index access vs. table access

As we can see we have a lot more I/O requests on the table access than on the index access. To understand this we have to think about how the buffer cache works. The index has 3075 leaf blocks and the table has 78954 blocks.

Each segment is accessed 175000 times (see figure 1). For the index that means 57 hits per leaf block. For the table that would mean roughly two hits per block.

In addition the index is sorted based on the search criteria, therefore we can expect that not the whole index is retrieved, but only the part that is relevant for the search.

Thus under normal circumstances most of the 175000 buffer gets of the index access will be found in the buffer cache. In comparison a higher proportion of the buffer gets of the table access will translate into physical reads.

To add an additional filter column in den Index will therefore reduce the overall number of physical reads.

A long running query

Some years ago I got a query to tune that keep me thinking for years. Of course I did not work on it all the time, but occasionally it popped up and I started contemplating again how it could be solved.

On the first glance it seems very simple. In the essence it was "find all people with the name Miller in the city of Bern."

That seems very common, simple query. On a second glance it is not that simple. Miller (actually Müller in German) is a common name in Switzerland (as well as in Germany). Bern, the Swiss capital has about 350.000 inhabitants.

No matter where you start searching, you first result tends to be rather big, the overlapping final result is comparatively small.

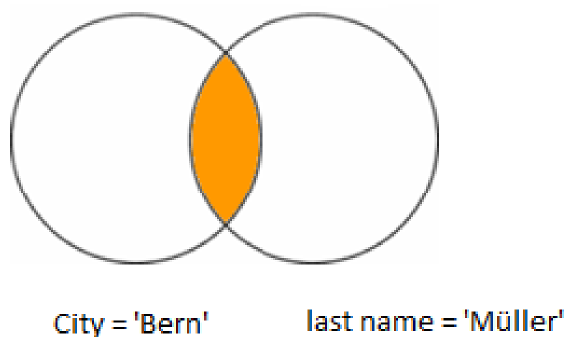

Fig. 3: Set diagram for the Miller in Bern search

Actually the best way would be to combine the two search criteria. That is hard to do, however, since the two search criteria are against different tables: one on table person, the other on table address.

Let's explore the different options to combine two search criteria in one index using features of the Oracle database.

Our underlying assumption is that we work in an OLTP environment, like a call center application. Actually that is the environment that typically generates such queries.

Most of our ideas would also work in a data warehouse too, but here the alternative solutions (see below) could be considered more seriously.

Materialized View

One way to deal with this situation would be to join table address and table person by building a materialized view and then build a composite index on top of it.

In order for this to work a few prerequisites must work out:

- creating a materialized view must be possible in the sense that we got the rights and the space etc.

- In an OLTP environment we must create a fast refresh on commit materialized view which can be very ambitious to manage. We are violating normal forms which will lead into update anomalies which can make the refresh very resource consuming.

This solution can work out in some carefully selected scenarios, but it is certainly not a generic solution.

Bitmap Join Index

In principle this solution has about the same drawbacks as the materialized view. It also holds a more serious drawback in the lock that will be applied due to the bitmap index.

In an OLTP environment this is not an option.

Text Index

It is possible to have a multi column, multi table text index. I have not much experience with text indexes and their limits in an OLTP environment. In any case Oracle Text Option would cost extra.

In addition we would have to change the queries to adopt text index search operators.

Searching for a Solution

I came across this type of query occasionally in my professional career. After all it is just a typical query used in call centers. Every time somebody told me: “Our call center software is tuned to the max and will respond instantly”. I told them to search for “Müller” in “Bern”, or for “Meier” in “Zurich”.

The result was always a jaw dropping long wait. After contemplating a while over the issue I came up with the idea that the query was indeed dominated by the problem of throw-away rows, or throw-away dominated, as I’ll refer to it going forward.

Let us look at figure 3 again. Only the orange Part is the Result. The rest is throw-away.

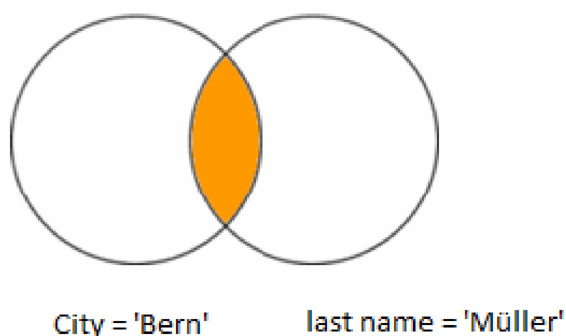

Fig. 4: Set diagram for the Miller in Bern search revisited

Actually the throw-away is far bigger than the result. So that means the query is throw-away dominated.

The question I am asking myself was “why would you go through the extra effort of reading a row from the table when you know with high probability that you are going to throw it away anyway?

The idea of the salted banana arrived in my head as a kind of metaphor for the table access that might now be required. Again, if you want to throw away your banana, why would you go through the process of peeling and salting it first?

Our Example

To redo and test the issues with the "Müller/Bern" query I have created a little example that does roughly resemble the real data set.

I created a schema containing two tables.

```
Create table address (city_name      varchar2(30) not null,
             person_id               number(7) not null,
             data                    varchar2(200))
/
create Table person (person_id number(7) not null,
                    last_name varchar2(15) not null,
                    data        varchar2(200))
/
```

As you can see the schema is simplified. It does not mean to be realistic, but be enough to aid our discussion.

In addition these indexes exist to allow various type of join sequences and to give the optimizer a choice.

```
create Index city_idx1 on address (city_name, person_id) compress;
create Index city_idx2 on address (person_id, city_name) compress;
create Index person_idx1  on person(last_name, person_id) compress;
create Index person_idx2  on person(person_id, last_name) compress;
```

I also created statistics. Please note that table person does have histograms while table address does not.
This is now the query we are discussing. The assumption is that it does run in an OLTP system, therefore we like to see the first ten rows only.

```
select * from (
select a.city_name, p.last_name, substr(p.data,1,1) p_data,
substr(a.data,1,1) a_data
  from ibj.address a,
       ibj.person p
where p.person_id = a.person_id
  and a.city_name = 'Bern'
  and p.last_name = 'Müller')
where rownum < 11
/
```

Execution plans

The classical execution plan would be a nested loop join.

```
------------------------------------------------------------------
| Id  | Operation                        | Name          |
------------------------------------------------------------------
|   0 | SELECT STATEMENT                 |               |
|*  1 |  COUNT STOPKEY                   |               |
|   2 |   NESTED LOOPS                   |               |
|   3 |    TABLE ACCESS BY INDEX ROWID   | ADDRESS       |
|*  4 |     INDEX RANGE SCAN             | CITY_IDX1     |
|   5 |    TABLE ACCESS BY INDEX ROWID   | PERSON        |
|*  6 |     INDEX RANGE SCAN             | PERSON_IDX2   |
------------------------------------------------------------------
```

Fig. 5: traditional execution plan

As we already know, the plan has its weakness in the excessive time it takes while executing against both tables.

Of course Oracle has identified the issue and has started taking measures against it.

As Randolf Geist states:

Oracle introduced in recent releases various enhancements in that area - in particular in 9i the "Table Prefetching" and in 11g the Nested Loop Join Batching using "Vector/Batched I/O".

The intention of Prefetching and Batching seems to be the same - they both are targeted towards the usually most expensive part of the Nested Loop Join: The random table lookup as part of the inner row source. By trying to "prefetch" or "batch" physical I/O operations caused by this random block access Oracle attempts to minimize the I/O waits. [3]

In the Table prefetch the access to the non driving table is done outside of the nested loop, which is shown in the figure below. The rowids are collected and buffered inside the nested loop. For adjacent blocks, the table access by rowid is done as a small vector I/O.

```
------------------------------------------------------------------
| Id  | Operation                        | Name          |
------------------------------------------------------------------
|   0 | SELECT STATEMENT                 |               |
|*  1 |  COUNT STOPKEY                   |               |
|   2 |   TABLE ACCESS BY INDEX ROWID    | PERSON        |
|   3 |    NESTED LOOPS                  |               |
|   4 |     TABLE ACCESS BY INDEX ROWID  | ADDRESS       |
|*  5 |      INDEX RANGE SCAN            | CITY_IDX1     |
|*  6 |     INDEX RANGE SCAN             | PERSON_IDX2   |
------------------------------------------------------------------
```

Fig. 6: Table prefetching

Nested Loop batching goes one step further, as the access to the outer tables is done as a complete separate step.

As stated in [1], paragraph 11.6.3.1.2:

Oracle Database 11g Release 1 (11.1) introduces a new implementation for nested loop joins to reduce overall latency for physical I/O. When an index or a table block is not in the buffer cache and is needed to process the join, a physical I/O is required. In Oracle Database 11g Release 1 (11.1), Oracle Database can batch multiple physical I/O requests and process them using a vector I/O instead of processing them one at a time. As part of the new implementation for nested loop joins, two `NESTED LOOPS` *join row sources might appear in the execution plan where only one would have appeared in prior releases. In such cases, Oracle Database allocates one* `NESTED LOOPS` *join row source to join the values from the table on the outer side of the join with the index on the inner side. A second row source is allocated to join the result of the first join, which includes the rowids stored in the index, with the table on the inner side of the join.*

```
-----------------------------------------------------------------
| Id  | Operation                        | Name
-----------------------------------------------------------------
|   0 | SELECT STATEMENT                 |
|*  1 |  COUNT STOPKEY                   |
|   2 |   NESTED LOOPS                   |
|   3 |    NESTED LOOPS                  |
|   4 |     TABLE ACCESS BY INDEX ROWID  | ADDRESS
|*  5 |      INDEX RANGE SCAN            | CITY_IDX1
|*  6 |     INDEX RANGE SCAN             | PERSON_IDX2
|   7 |    TABLE ACCESS BY INDEX ROWID   | PERSON
-----------------------------------------------------------------
```

Fig. 7: nested loop batching

As you will see our proposed solution (below), which we call an “index backbone join”, follows that same path in a more generic manner.

But first, let us have a close look on the runtime statistics of the plan chosen by the optimizer. The optimizer uses nested loop batching.

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		1		10	00:03:24.42	44271	29509
* 1	COUNT STOPKEY		1		10	00:03:24.42	44271	29509
2	NESTED LOOPS		1		10	00:03:24.42	44271	29509
3	NESTED LOOPS		1	11	10	00:03:24.31	44261	29499
4	TABLE ACCESS BY INDEX ROWID	ADDRESS	1	12	14731	00:01:55.73	14793	14792
* 5	INDEX RANGE SCAN	CITY_IDX1	1	44244	14731	00:00:00.93	62	61
* 6	INDEX RANGE SCAN	PERSON_IDX2	14731	1	10	00:01:28.51	29468	14707
7	TABLE ACCESS BY INDEX ROWID	PERSON	10	1	10	00:00:00.11	10	10

Fig. 8: runtime statistics for nested loop batching

As we can clearly see, the access to table Address takes most of the time. So why would we access the table, when the row we get will likely be thrown away? All the necessary information to select and join the rows will be found in the index already.

The index backbone join consist of two phases: in the first phase only the indexes are used to find the result set rowids. When we have the result set rowid, we do a delayed table access for **both** tables.

That is the major difference to nested loop batching. Here the access to **one** table is delayed.

The optimizer is not capable of doing an index backbone join, but we can simulate it with a manual rewrite. Note that we need a no merge hint to separate the index only scans from the table access by rowid phase.

```
select * from (
select a.city_name, p.last_name, substr(p.data,1,1) p_data,
       substr(a.data,1,1) a_data
  from ibj.person p,
       ibj.address a,
       (select /*+ NO_MERGE  */ p.rowid p_rowid, - phase 1
               a.rowid a_rowid
           from ibj.address a,
                ibj.person p
           where p.person_id = a.person_id
            and a.city_name = 'Bern'
            and p.last_name = 'Müller'
         ) i
where p_rowid = p.rowid
  and a_rowid = a.rowid)
where rownum < 11
/
```

Before each run, the buffer cache got flushed. Therefore the results are compareable.

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		1		10	00:00:00.40	150	149
* 1	COUNT STOPKEY		1		10	00:00:00.40	150	149
2	NESTED LOOPS		1	10	10	00:00:00.40	150	149
3	NESTED LOOPS		1	10	10	00:00:00.26	140	139
4	VIEW		1	42105	10	00:00:00.14	130	129
* 5	HASH JOIN		1	42105	10	00:00:00.14	130	129
* 6	INDEX RANGE SCAN	PERSON_IDX1	1	42105	15445	00:00:00.01	68	68
* 7	INDEX RANGE SCAN	CITY_IDX1	1	44244	14731	00:00:00.11	62	61
8	TABLE ACCESS BY USER ROWID	PERSON	10	1	10	00:00:00.13	10	10
9	TABLE ACCESS BY USER ROWID	ADDRESS	10	1	10	00:00:00.14	10	10

Fig. 9: runtime statistics for simulated index backbone join

Although the estimates are wrong, the execution plan is robust.

Conclusion

The idea of the index backbone join brought a different view on execution plans.

There are columns which are essential for a plan. For example those used in the where clause and those used in joins.

In this context we have been looking at a specific example combining two tables.

However it is easy to imagine a more generic solution combining as many indexes needed in a first phase to build the essential part of the query, building the backbone of the execution plan. Therefore the name *index backbone join*.

Certainly a prerequisite is specific indexing. There is no much drawback in this approach, because there are no additional indexes needed, but rather extra columns attributed to existing indexes.

The gain could be faster and more robust executions plans.

Acknowledgment

I like to thank my friend and co-inventor Björn Engsig who holds the US Patent 2011/0119249 A1 together with me. Björn has contributed valuable improvements to the concept of the index backbone join. His sharp mind and his clear sight was much appreciated.

I also like to thank Tim Gorman and Randolf Geist for their kind and professional review. All the flaws of this document are mine, not theirs of course.

As Randolf pointed out Jonathan Lewis has documented a similar idea in his blog [4]. It was Jonathan Lewis's great fan hit ratio presentation that inspired to the title of this text.

References:

[1] Oracle® Database Performance Tuning Guide
[2] http://afatkulin.blogspot.ch/2009/01/consistent-gets-from-cache-fastpath.html
[3] http://oracle-randolf.blogspot.ch/2011/07/logical-io-evolution-part-2-9i-10g.html
[4] http://jonathanlewis.wordpress.com/2010/05/18/double-trouble/

Contact:
Lothar Flatz
Centris AG
Grabackerstr., 2
CH-4502 Solothurn

Telefon:	+41 (0) 32-625 43 48
E-Mail	Lothar.Flatz@Centrisag.ch
Internet:	www.centrisag.ch