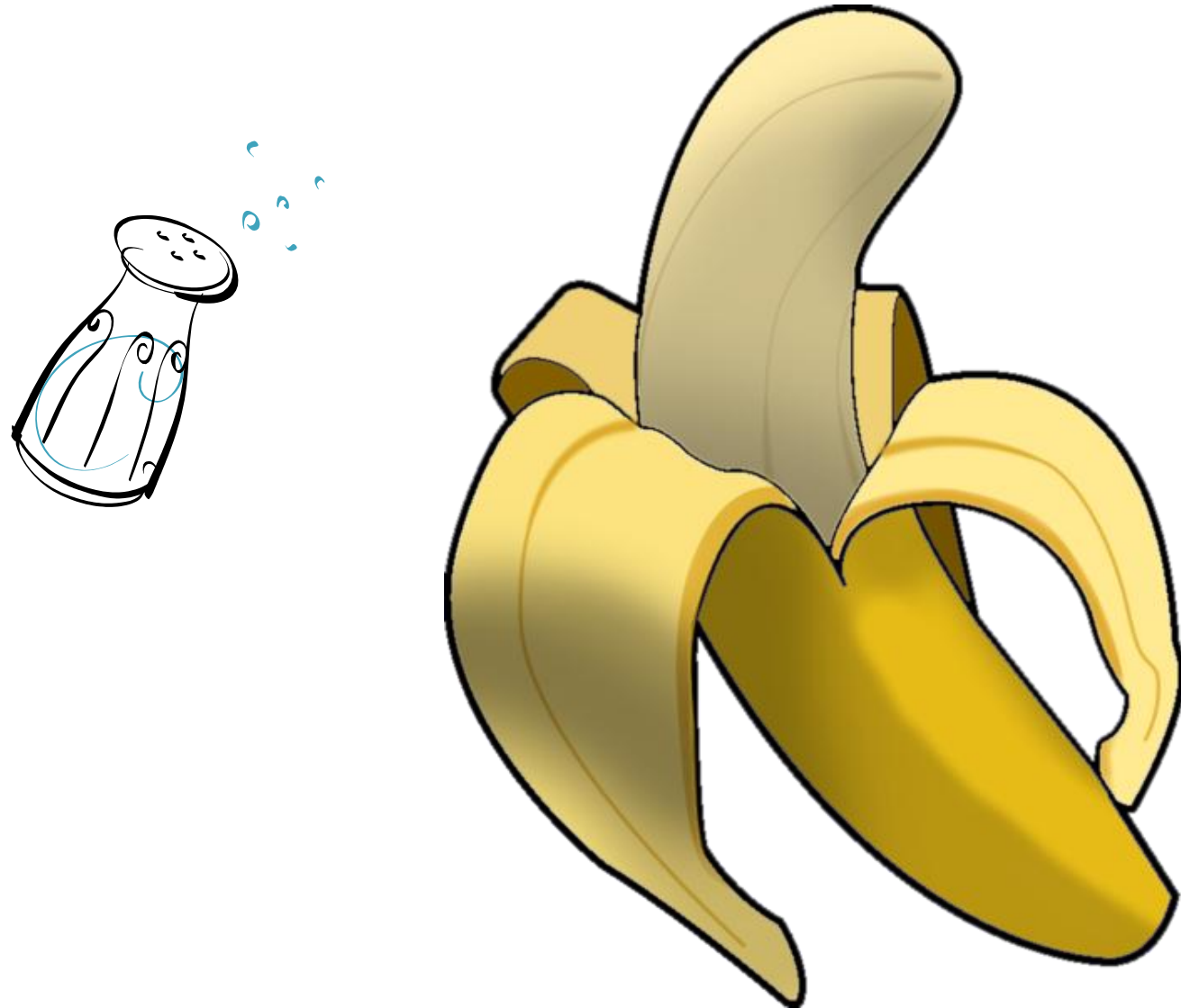


How to avoid a salted Banana

Lothar Flatz
Centris AG
Solothurn

A Joke that highlights an issue



The concept of throw away in performance tuning

- commonly used in Performance tuning
- usually in the context of Join Sequence optimization
- or index optimization

| 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 |

- all but one row thrown away

To cache an (B-tree) index versus a table

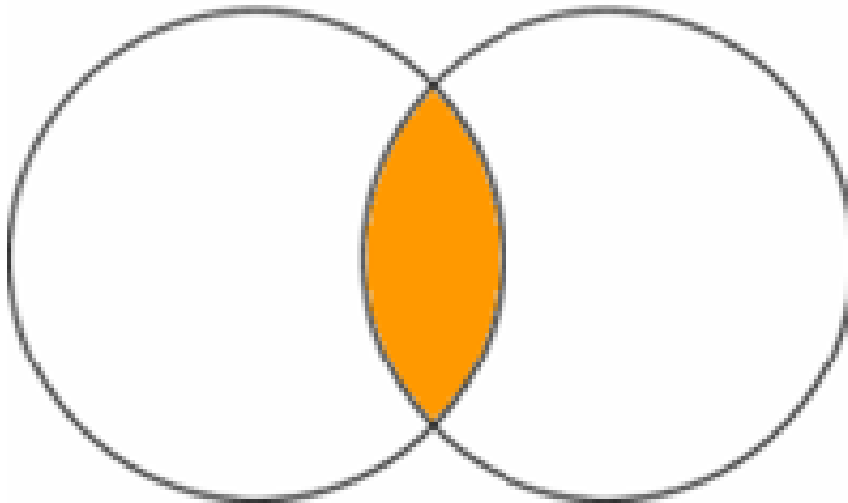
- generally indexes are cached better than underlying tables
- index blocks are likely to have a better hit ration than table blocks
 - index entries are usually smaller than underlying rows
 - indexes are clustered by definition

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

- the table access generates 19 times more I/O than the index access

A challenging query

- common query in call centers
- No matter where you start you always get a high number of rows
- the intersection of the two criteria is small
- query is throw away dominated



City = 'Bern'

last name = 'Müller'

try to combine the criteria

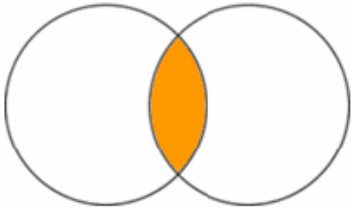
- natural approach is combined search
- main issue is that the search criteria are on different tables
- No easy way to create combined search



different traditional approaches

- fast refresh materialized view
 - need rights and additional space
 - fast refresh very critical in OLTP
- bitmap join index
 - locking implications
 - fast refresh very critical in OLTP
- text index
 - “exotic” solution
 - requires purchase of text option
 - program change required for search operators

Why read the table if you are going to throw it away?



City = 'Bern'

last name = 'Müller'



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

=



example code

```
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))

/

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;
```

example code continued

```
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  
/
```

Oracle Optimizations for inner Table access

Randolf Geist (<http://oracle-randolf.blogspot.ch/2011/07/logical-io-evolution-part-2-9i-10g.html>):

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.

table prefetching (9i)

- access to inner table outside of the nested loop
- rowids are collected and buffered inside the nested loop
- access by rowid is done by 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 |

nested loop batching (11g)

- extra nested loop introduced
- rowids are collected and buffered inside the nested loop
- access by rowid is done by a big vector I/O

| 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 |

example code rewritten for Index Backbone Join

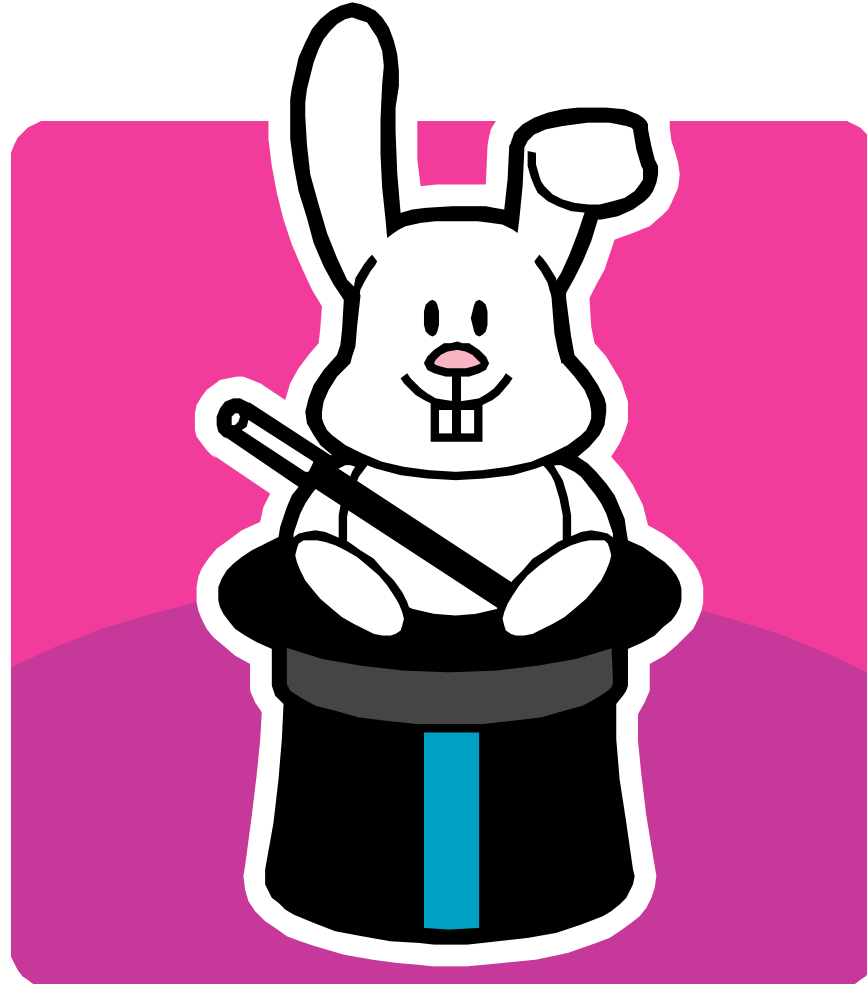
```
select * from (  
select a.city_name, p.last_name, substr(p.data,1,1) p_data,  
       substr(a.data,1,1) a_data  
from ibj.person p,  
     ibj.address a,  
     (select /*+ NO_MERGE */ p.rowid p_rowid,  
          a.rowid a_rowid  
       from ibj.address a,  
            ibj.person p  
      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

Index Backbone Join

- extra nested loop programmed
- all filtering in the indexes
- rowids are cached in the intermediate resultset
- access to the tables is delayed for the last possible moment

| Id | Operation | Name |
|-----|----------------------------|-------------|
| 0 | SELECT STATEMENT | |
| * 1 | COUNT STOPKEY | |
| 2 | NESTED LOOPS | |
| 3 | NESTED LOOPS | |
| 4 | VIEW | |
| * 5 | HASH JOIN | |
| * 6 | INDEX RANGE SCAN | PERSON_IDX1 |
| * 7 | INDEX RANGE SCAN | CITY_IDX1 |
| 8 | TABLE ACCESS BY USER ROWID | PERSON |
| 9 | TABLE ACCESS BY USER ROWID | ADDRESS |

Demo



Timings – native 11G

- Optimizer decides for nexted loop batching
- table access for Address almost 2 minutes
- batched access to person table Person fast

| 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 |

Timings – Index Backbone Join

- no_merge hint necessary or query gets rewritten to previous result
- non optimized view allows for hash join
- very efficient
- buffer cache was flushed for fair comparison

| 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 |

Conclusions

- completely different look on queries

```
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,  
        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 i.p_rowid = p.rowid  
        and i.a_rowid = a.rowid)  
where rownum < 11
```

Ballast



essential
columns



Quo vadis ?

