Warm Cache Costing – A Feedback Optimization Technique for Buffer Pool Aware Costing

Ramanujam H S  
Sybase Software India Pvt Ltd, Pune  
ramanujam.s@sybase.com

Edwin Seputis  
Sybase Inc, Dublin, CA  
edwin.seputis@sybase.com

ABSTRACT
Most modern RDBMS depend on the query processing optimizer’s cost model to choose the best execution plan for a given query. Since the physical IO (PIO) is a costly operation to execute, it naturally has an important weight in RDBMS classical cost models, which assume that the data is disk-resident and does not fit in the available main memory. However, this assumption is no longer true with the advent of cheap large main memories.

In this paper, we discuss the importance of considering the buffer-cache occupancy during optimization and propose the Warm Cache Costing (WCC) model as a new technique for buffer-pool aware query optimization. The WCC-model is a novel feedback optimization technique, based on the execution statistics by learning PIO-compensation (PIOC) factors. The PIOC factor defines the average percentage of a table which is cached in the buffer pool. These PIOC factors are used in subsequent query optimizations to better estimate the PIO, thus leading to better plans.

These techniques have been implemented in Sybase Adaptive Server Enterprise (ASE) database system. We have observed that they provide considerable improvements in query timings, in Decision Support environments and with almost negligible regression(if any) in other environments. This model enjoys the advantage of requiring no change to the buffer manager or other modules underlying the query processing layer and therefore is easy to implement. Also, since this model is part of an extensive feedback optimization architecture, other techniques using feedback optimization framework can be plugged in easily.

1. INTRODUCTION
Most modern query optimizers for relational database management systems(RDBMS) determine the best query execution plan for executing a SQL query by mathematically modeling the execution cost for each plan and choosing the valid cheapest plan.

The optimizer explores a number of candidate plans in the search space, estimating each of the plan’s cost using a cost model and picking the cheapest according to that model [15]. Though this basic approach to query optimization has not changed, the rest of the environment in which the databases operate has changed dramatically. Processors have become exceedingly faster and memories have become many times bigger.

The ability to execute increasingly complex queries over very large database(VLDB) environments has increased at a higher rate than the ability to optimize such complex queries. Query optimization has again become a subject of research with more emphasis on modeling optimizers to the dynamic nature of the databases. One of the techniques, is the dynamic query optimization in which optimization and execution are interleaved [12] using similar techniques such as choose operators [6], etc.

Optimizers commonly assume that all the base tables involved in the query are disk-resident. Even though this assumption is generally safe, main memories have grown much and those plans which are considered to be cheap might not actually be so, if the base tables(or part of them) are actually available in the main memory as shown in Section 2. Ignoring the contents of the buffer pool while optimizing queries can cause the optimizer to pick up sub-optimal plans.

Example 1:

SELECT l_orderkey, l_quantity, l_extendedprice  
FROM lineitem  
WHERE l_shipdate < ‘2004/01/01’

For example, if we consider the above simple query, the optimizer has lot of options to evaluate this query, viz:

- use non clustered(NC) index – LINEITEM.Shipdate defined on l_shipdate to filter the rows based on index key and do a data page lookup for other columns
- use NC index – LINEITEM.Suppart to do a sequential scan of all the index pages(only the index pages since it is a covered index 1)
- use a table scan on the LINEITEM table

Generally beyond a threshold value in predicate selectivity, optimizers will almost always pick the table scan. However, if the data

1An index covers a query if it contains all the columns needed to answer the query as part of the index key itself and therefore no data page lookup is necessary
pages that are actually accessed when evaluating the predicate are already in the buffer cache, using index scan might be faster. Data cached in the buffer cache might also affect other choices made during query optimization including join ordering and the selection of join algorithm. Therefore enabling query optimizers with the cognizance of cache contents might help DBMS in selecting much better plans.

There has been earlier research that discusses the advantages of considering the cache contents while optimizing the queries [14]. This work clearly argues the advantages of the having the cache cognizance when optimizing queries.

The approach followed in [14] gives fairly exact predictions of cache contents, since it queries buffer manager to find out the cache occupancy information when optimizing a query. But, we believe that the cost of obtaining such exact information and the changes required at the buffer manager level for providing such information are considerably high.

Also it is possible that correlations between data may not be known, therefore an access path which may seem to have poor cache hits with many PIOs assuming random independent events, might actually has very good cache hits with few PIOs for a respective index scan.

On the other hand, we believe that cost models that reflect the buffer contents are likely to make a difference when the workload has some definite locality. Gray and Graefe in their five-minute rule analysis [10, 9], identify that it makes economic sense to make sure pages referenced every five minutes are memory resident. Graefe in [7] argues that the five-minute rule still holds true but for large pages appropriate to today’s disks and small pages with respect to fast transfer bandwidth flash disks.

Therefore, if we assume that the cache sizes for the databases are tuned to follow the above stated five-minute rule, then we feel that considering approximate cache occupancy information is a good enough measure, especially if such information could be derived with least cost and almost non-intrusive approach.

In this effort, we propose a novel usage of execution metrics captured as part of the current query and enable the costing module to use this learned/cache occupancy information in optimizing future queries. Our techniques are easy to implement since they require no changes to buffer manager or other modules underlying the query processing layer. Also, overheads introduced by these techniques are minimal.

In the following sections we describe our technique in general with specific details of how it is adapted for Sybase ASE. Though the technique is detailed with respect to Sybase ASE, it can be noted that the model is generic and is applicable to any DBMS in general.

The rest of the paper is organized as follows: Section 2 introduces the related previous work in this area. Section 3 discusses our basic approach and implementation details. Section 4 illustrates the maintenance overhead of our approach. Section 5 details the experimental evaluation of our implementation. Section 6 presents the extensions and future directions and Section 7 concludes our work.

2. RELATED WORK

Our work is based on many prior research and development efforts, which we survey now in some depth because our design is primarily a novel combination of those existing techniques.

2.1 Importance of Cache Cognizant Optimizer

This section examines how the cache contents of the buffer pool can affect the access path selection. We briefly discuss the idea here, while the original paper [14] could be referred for complete information.

Query optimizers use cost functions to evaluate alternate evaluation plans. In choosing between a table scan and a NC index scan, this translates to some cut-off in predicate selectivity. For predicates more selective than that cut-off, the optimizer chooses a NC-index scan, while for predicates with less selectivity a table scan is chosen. An interesting point to note is that this threshold value is independent of the contents of the buffer pool. Over the course of executing the queries, a significant number of pages that are required to answer a query might have become cached in the buffer-pool. The relative performance of the alternative query plans can change based on how many pages are resident in the buffer pool.

This paper defines an analytical model which defines that theoretically there is a positive range of predicate selectivity where the traditional optimizers might pick a bad plan as the best plan. We briefly touch upon the model below, since it forms the basis for our work too.

2.1.1 Analytical model

Consider a relation R and a hypothetical workload, which consists of queries with selection predicates on R. Assume that R has NC index defined on the attributes involved in the predicate. In such a situation, the optimizer selects the cheapest plan from 2:

- $TableScan(cost : N \times T_{seq})$
- $IndexScan(cost : C(k) \times T_{random})$

Where, $N$ – number of pages in R, $N_{buf}$ – size of buffer cache, $T_{seq}$ – sequential IO time, $T_{random}$ – random IO time and

$$C(k) = N \times (1 - (1 - 1/N)^k)$$

Where $C(k)$ represents cardenas formula [4, 18] for estimating the number of pages touched while accessing k records using a NC index. Using the above equation, the cut-off value in terms of number of records after which index scan becomes costlier is:

$$N \times T_{seq} = C(k) \times T_{random}$$

$$N \times d = N \times (1 - (1 - 1/N)^k)$$

$$k_0 = \log(1-d)/\log(1-1/N)$$

Where, $d = T_{seq} / T_{random}$. Similarly, if $f$ – represents the fraction of pages already cached in the buffer pool then,

2 Assuming a simplistic cost model that considers only IO cost for selecting best plan.
3. WARM CACHE COSTING MODEL

- TableScan\(\text{(cost : } (N - f \times C(k)) \times T_{\text{seq}})\)
- IndexScan\(\text{(cost : } C(k) \times (1 - f) \times T_{\text{random}})\)

Simplifying which leads to

\[ k_1 = \log((1 - d) \times (1 - N_{\text{buf}}/N))/\log(1 - 1/N) \]

Thus \(k_0, k_1\) are the cut-off predicate selectivity values in terms of number of records, after which traditional optimizer and buffer pool aware optimizer respectively, would pick TableScan for evaluating the query. Therefore,

\[ k_1 - k_0 = \log(1 - N_{\text{buf}}/N)/\log(1 - 1/N) \]

\'\(k_1 - k_0\)’ represents a positive selectivity range in which a buffer pool aware optimizer would pick an IndexScan while a traditional optimizer would wrongly pick a TableScan.

2.2 Feedback using Execution Metrics

While query optimizers do a remarkably good job of estimating the cost of most queries, many assumptions underlie this mathematical model like independence of predicates, currency of information, etc. Most of the DBMS try to address these problems using histograms, multi-attribute densities, etc.

There has been research in the recent past on validating the estimations of the query optimizers using actual statistics collected during the execution of the query. Most of this research was concentrated on row count estimation [16], distinct page count estimation [5], self-tuning histograms [3], etc.

In [3] the authors present a query feedback loop, in which actual cardinalities gleaned from executing a query are used to correct histograms. Multiple predicates can be used to detect correlation and multi-dimensional histograms. This paper proposes effective techniques for single-table predicates, but doesn’t deal with join predicates, aggregations, etc. LEO, DB2’s Learning Optimizer [16] generalizes and extends this work to all predicates and any operators. LEO incorporates a comprehensive technique for a query optimizer to actually learn from any modeling mistake in an execution plan by automatically validating its estimates against actuals, thus determining at what point in the plan significant errors occurred and adjusting the model dynamically to better optimize the future queries.

3. WARM CACHE COSTING MODEL

In classical RDBMS cost models, the cost of an operator is a function of number of rows processed, the number of IOs spent, and the cost of CPU operations processed, etc.

When an ad hoc query is being evaluated, the costing module estimates Logical IO(LIO), using the statistics available on the object. Then depending on the number of times the object is accessed, type of the access path, size of the buffer cache available, etc. its Physical IO PIO is calculated \(^3\). These are referred as estimatedLIO and estimatedPIO respectively in WCC-model.

Once these different parameters of the cost function are estimated the cost of the operator is calculated, alternative plans for the query are evaluated and the valid cheapest plan is selected.

Thus selected plan (called as Best Plan) is passed to the execution engine, which after executing it, captures the actual execution statistics that the plan incurs. The number of IOs spent actually by the execution engine are noted as actual LIO and actual PIO.

This is a wealth of information and can be used to understand and automate a lot of design and statistic errors in the optimizer [16].

In this paper, we consider understanding the dynamic nature of the buffer cache using these execution metrics.

In this effort, we propose a new costing model called the Warm Cache Costing model. The WCC model is an extensive feedback mechanism developed using statistics from execution engine.

Under WCC model, after the estimatedPIO for the object is calculated, the PIO Compensation factor(PIOC)[described in Section 3.1] for that object is obtained from the catalogs (referred to as oldPIOC) and is applied to obtain the compensatedPIO.

Once the compensatedPIO is obtained, the costing of the scan operator and further operators are done using the compensatedPIO, instead of estimatedPIO.

As part of the execution of the query, the execution engine, captures actualLIOs/actualPIOs, i.e., the LIOs/PIOs actually spent in answering the query. The corresponding estimated values are propagated on from the costing module to the execution engine.

These estimated and actual LIOs/PIOs are used to calculate newPIOC, as a function of the passed in values and the oldPIOC. This newPIOC is used in the optimization of future queries on the object.

3.1 PIO Compensation Factors

Since PIO is a costly operation, it naturally has an important weight in the underlying cost models. Also since the cost models assume that the object is disk-resident, it is possible for the actualPIO and the estimatedPIO to be different. This difference is captured as PIO Compensation Factor(PIOC):

\[ \text{PIOC} = \frac{\text{actualPIO}}{\text{estimatedPIO}} \]

In this equation it is possible that there is a difference in the estimatedPIO and the actualPIO not just because of availability of the object in the cache, but because of stale/missing statistics. It is also possible that the mis-estimation is because of incorrect row count estimation of the objects preceding in the join order. Estimating the row count or cardinality after one or more predicates have been applied, has been the subject of research for over 20 years [15, 17].

Especially, it is very difficult to accurately predict the join cardinality estimation after 1-2 join operators [11].

It can be seen from Figure 1 that though the estimations of row count with respect to both the individual predicates are correct, the estimation of the row count on LINEITEM table when we have both the predicates is incorrect (as shown in Figure 2). This is because

\(^3\)Every IO issued by the system is considered a logical IO, while only those logical IOs, which incur a cache-miss and does an IO to the secondary storage system is considered as a physical IO.
we assume that the predicates are independent, but here it turns out that they are not. Though, this could be solved if we have multi-attribute histograms on the pair of columns involved in the predicates\(^3\), what we are trying to depict is that mismatch in estimation can happen in row count, LIO and thus PIO, not just because of availability of the table in cache but because of other inconsistencies too.

Therefore it is important that our PIO model ignores the mis-estimations because of such inconsistencies so that the model only caters to PIO difference because of cache occupancy. To avoid such problems, we calculate unit PIO mis-estimation by proportionating the estimatedPIO with estimatedLIO

\[
\text{estimatedPIO} = \frac{\text{estimatedPIO}}{\text{estimatedLIO}}
\]

Similarly,

\[
\text{actualPIO} = \frac{\text{actualPIO}}{\text{actualLIO}}
\]

Therefore, PIO or rather current PIO(currPIO), is calculated using these proportionated actual and estimated values, so that it doesn’t suffer from any of the above stated inconsistencies.

\[
\text{currPIO} = \frac{\text{actualPIO}}{\text{estimatedPIO}}
\]

This currPIO captures the mis-estimation in PIO, for the current access of the object. It might not be safe to calculate newPIO only basing on this currPIO, because it is possible that a very cold object comes into the cache transiently and it should not be considered hot. To avoid such wild swings in PIO values, it is averaged over \(p\) iterations. We use a modified Exponential Moving Average(EMA) function:

\[
\text{newPIO} = \left(\frac{p - q}{p}\right)\text{oldPIO} + q \times \text{currPIO}
\]

Where,

- \(p\) – PIO factor window
- \(q\) – oldPIO – PIO for previous iterations
- currPIO – PIO for the current iteration

\(p, q\) control the relative importance to be given to oldPIO and the currPIO. By tuning these values, we can control the PIO learning to be more oriented to the current access of the object or to give equal importance to the current access as that of old queries in the history.

Thus, PIO factors capture the average PIO mis-estimation, by the costing module for the given object, over previous \(p\) queries involving that object. Thus, compensatedPIO represents the actual PIOs that the object might do, considering the PIO mis-estimation over the previous \(p\) accesses of the object.

### 3.2 Adapting WCC to Sybase ASE

The ASE Optimizer Costing module, is a bottom-up costing model, similar to the Volcano Optimizer model proposed by Graefe [8]. In the model, the cost of an operator is a function of number of Logical IOs(LIOs), number of Physical IOs (PIOs) and the cost of CPU operations over the rows processed [1].

The statistics module of the ASE captures the many different metrics as part of execution of a given query. An example of the statistics captured by ASE is shown in Figure 2. In Figure 2, value of \(r\): and \(\varepsilon\): correspond to actual and estimated rows. Similarly, value of \(l\), \(\varepsilon\) and \(p\), \(\varepsilon\): correspond to the actual and estimated values of LIO and PIO respectively.

As part of implementation of WCC model in Sybase ASE these estimated and actual IOs are used to calculate currPIO as described in Section 3.1. Apart from that Sybase ASE allows a DBA to configure any number of buffer caches and each of them can be configured as a set of up to 4 pools, each with different page size(called the mass size of the pool) viz., 2, 4, 8, 16 times the server page size.

It is also possible for the DBA to configure each of these pools to be of different sizes. Each table/index can be bound to a different cache and the optimizer uses information like the number of pages of the object being accessed, how clustered the pages are, etc. to determine whether to use a large mass pool(using prefetch) or to use a small mass pool to access the object. The optimizer passes these hints to the buffer manager, to choose the pool, for accessing the object.

\(^3\)Sybase ASE doesn’t allow having multi-attribute histograms. It maintains multi-attribute densities modeled similar to [13].
To take these different pools into consideration, we ensure that there are two different PIOC factors maintained one each for large and small mass pools, i.e., largeMassPIOC & smallMassPIOC.

**Example 2:**

```
SELECT *
FROM lineitem
WHERE l_shipdate > '2004/01/01'
```

In the above query, If we assume that there is a NC index on `l_shipdate` of `LINEITEM` table, then the index pages are traversed and the qualifying RIDs are collected to make corresponding data page lookup to answer the query. Therefore, it is possible that among the total actualPIOs few PIos are made for index pages and few for data pages. For performance reasons, databases are tuned to make sure that index pages are always available in the cache. In ASE, it is possible to bind the index and data pages of an object to different caches/pools. Therefore, DBAs tend to bind index and data pages to different caches to improve cache hits and there by performance.

Therefore, it is important for the WCC model to take care of this and maintain different PIOC values for each of these pools. If not, it might be possible that the cache behavior predictions by the PIOC values are incorrect.

For example, Let us assume that the estimatedPIO for the index scan is 300, where the distribution of (indexPIOs, dataPIOs) is (100, 200), while actualPIOs were only 100, distributed as (0, 100) i.e., no PIos for index pages and 100 PIos for data pages. If the distribution of the pages is not considered, it is possible that the PIOC factor will predict that 33% of the index and data pages are cached, while in reality, 100% of the index pages and 50% of the data pages are cached.

Therefore, to take care of this distribution, PIOC factors are maintained separately for index and data caches. This leads us to two more PIOC factors. Therefore, on the whole, we maintain four different PIOC factors for an object, viz.,

1. largeMassDataPIOC
2. largeMassIndexPIOC
3. smallMassDataPIOC
4. smallMassIndexPIOC

Each of these PIOC factors is maintained independently in the catalogs.

### 3.3 PIOC Granularity

It is possible for a set of queries which access only a part of the object to drive the PIOC towards 0, there by forcing the WCC model to assume that the object is hot.

For example, It might be possible that the `LINEITEM` table of the TPCH [2] database is semantically partitioned based on the value of `l_orderkey`, so that all the orders below a particular value are placed in a separate partition. In such a scenario, repeated execution of a query like the one specified in Example-3 might access the data from a single partition. Therefore repeated execution of this query(even with different values bounded to `@shipdate`) might access data only in that partition, there by making it hot. Because of this it is possible that the PIOC values for that table move towards 0.

Now, altogether if a different query, which is trying to access data from a different partition of the same table comes, PIOC model incorrectly predicts that even that partition is hot since the PIOC value for the whole table is 0. This problem was encountered because PIOC values were being maintained at table level.

**Example 3:**

```
SELECT *
FROM lineitem
WHERE l_orderkey < 500000 AND l_shipdate = '@shipdate'
```

To avoid such scenarios we reduce the granularity of the PIOC values to partition levels, so that accessing the pages of a partition will affect the PIOC values of only that partition and keep the PIOC values of the other partitions unchanged.

Frequently, applications create hot spots in tables such that the cache hits are very high. In those cases only a relatively small fraction of the table may be cached. In such scenarios, it should be possible for the PIOC factors to correctly predict the typical access pattern.

### 4. OVERHEAD OF PIOC MAINTENANCE

PIOC factors are maintained for `(object, partition, cache type, mass size)` combination in the system catalogs. Since these values are updated at the end of execution for each query, we tried to ensure that the overhead is minimal.

At the end of execution of every query we identify the PIO mis-estimation and the `currPIOC` is calculated accordingly using the formulae defined in Section 3.1. Using `oldPIOC` from the catalogs and the `currPIOC` calculated for this iteration, `newPIOC` is calculated and updated back.

The following experiment clearly validates that the cost of calculating the `newPIOC` and updating the catalogs with this new value is minimal.

Two database tables, one which is 365 way partitioned (`partTab`) and other which is single partitioned (`singleTab`) are created in ASE. The tables are loaded with 4000 rows each. As part of the experiment we run a query, with a WHERE clause predicate, which selects a set of rows from n partitions out of the 365 in the case of `partTab`. We ran the same query on the `singleTab` to measure the additional cost we are spending in updating the PIOC values of n partitions. Table 3 captures the time taken for evaluating the query with varying values for n.

It can be seen that the cost of evaluating the query on `partTab`, selecting 100 partitions out of 365 takes 50ms to complete, while the same query on the `singleTab` takes just 6ms to complete. Though the growth of the cost for updating PIOC values is linear with the increase of number of partitions, we found the cost to be quite high.

5 The number of rows is not significant for the experiment.
It was not very clear whether the extra cost is the cost of PIOC maintenance, or the initial setup cost before optimization, which reads catalog information for all the 100 partitions.

<table>
<thead>
<tr>
<th># of partitions</th>
<th>partTab</th>
<th>SingleTab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43</td>
<td>6</td>
</tr>
<tr>
<td>50</td>
<td>43</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>200</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>300</td>
<td>56</td>
<td>6</td>
</tr>
<tr>
<td>365</td>
<td>56</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 3: Time(in ms) for optimization and Execution

To verify that we repeated the experiment, this time noting only the execution cost for the query which includes the PIOC maintenance cost. Table 4 captures the execution cost for the above experiment. It can be seen from the table that the cost for execution of the query and the maintenance of PIOC values for 300 partitions is 6ms, while the cost for execution on a single partition table along with single PIOC update is 3ms. Though the 100% extra cost looks significant, it can be noted that this cost is constant irrespective of the number of rows being touched/accessed in each of the partitions, or the number of rows actually getting selected by the query.

<table>
<thead>
<tr>
<th># of partitions</th>
<th>partTab</th>
<th>SingleTab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>200</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>300</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>365</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4: Time(in ms) for only Execution

Therefore, this extra cost becomes insignificant as the execution time of the query increases. It can also be noted that the extra cost for PIOC maintenance is negligible in case of queries selecting data from less than 50/100 partitions, which is generally the case for most of the queries. If a query is accessing more than 300 partitions of the table, then probably it is a report generation query, which runs for quite some time and the extra cost of 3ms for PIOC maintenance will be acceptable, if the new plan selected by the WCC model, gives fair advantage in query execution time with respect to the old plan.

5. EXPERIMENTAL RESULTS

All experiments in this paper are performed on Sybase ASE database with 4KB server page size, on a 2GHz AMD Opteron 8 CPU box running Red Hat 3.4.6-2. The cache size of the server is tuned to be 800MB, distributed as 200MB for a pool with 4KB mass size and 600MB for a 32KB mass size pool. TPC-H [2] database of 1GB version is used for the experiments. The disk size of the whole database including the size of data, index and other meta data is about 4.5G.

The queries are fired as 300 sequential streams, each stream containing all the queries from TPC-H set of 22 queries in a random order. We used the qgen of TPC-H 2.8.0 standard specification, for the random stream generation.

Sybase ASE is extended with WCC model feedback optimization techniques and the execution module of ASE is extended to recalculate PIOC and store them back in catalogs, so that they are available for subsequent optimizations. Catalogs containing the PIOC values are cached in the memory, to avoid high PIOC maintenance costs. The values for $p$ - PIOC factor window and $q$ - PIOC factor weight discussed in Section 3, are taken to be 99 and 1 respectively.

Also each query is run in three modes:

- Default: Default optimization
- No PIO: Optimization assuming no PIO is spent on any object
- WCC: Optimization using WCC model

TPCH Query 16:

```sql
SELECT p_brand, p_type, p_size,
    count(distinct ps_suppkey) as supp_cnt
FROM part, partsupp
WHERE p_partkey = ps_partkey
  AND p_brand <> 'Brand#44'
  AND p_type NOT LIKE 'STANDARD ANODIZED'
  AND p_size IN (8, 46, 20, 19, 26, 41, 14, 18)
  AND ps_suppkey NOT IN ()
    SELECT s_suppkey
    FROM supplier
WHERE s_comment like '%Customer%Complaints%'
GROUP BY p_brand, p_type, p_size
ORDER BY supp_cnt DESC, p_brand, p_type, p_size
```

In Default mode, ASE consumes the elements of the or list into an in-memory table, and does a nested-loop join between this in-memory or-list and the index pi_psize as shown in Figure 5.

![Figure 5: Plan selected for query 16](image)

This is based on a cost based decision taken by the optimizer. It can be seen from the table that it is a better choice compared to doing a sequential scan on part table when the cache is cold. But when the cache is warm with the data pages of part table, the time taken for this plan is more than twice that of the sequential scan.

<table>
<thead>
<tr>
<th>Cache state</th>
<th>part_x</th>
<th>pi_psize</th>
</tr>
</thead>
<tbody>
<tr>
<td>cold</td>
<td>4633</td>
<td>4360</td>
</tr>
<tr>
<td>warm</td>
<td>1760</td>
<td>4170</td>
</tr>
</tbody>
</table>
While evaluating the query under WCC mode, over iterations, it was identified that table part is almost completely cached and therefore it is cheaper to do a sequential scan of part and restricting the rows by applying the predicates rather than using the or-list to probe the index. Figure 6 depicts the execution time taken for this query under different modes.

**TPCH Query 21:**

```sql
SELECT s_name, count(*) as numwait
FROM supplier, lineitem L1, orders, nation
WHERE s_suppkey = L1.l_suppkey
  AND o_orderkey = L1.l_orderkey
  AND o_orderstatus = 'F'
  AND n_name = 'FRANCE'
  AND s_nationkey = n_nationkey
  AND L1.l_receiptdate > L1.l_commitdate
  AND EXISTS (SELECT *
               FROM lineitem L2
               WHERE L2.l_orderkey = L1.l_orderkey
               AND L2.l_suppkey <> L1.l_suppkey )
  AND NOT EXISTS (SELECT *
                   FROM lineitem L3
                   WHERE L3.l_orderkey = L1.l_orderkey
                   AND L3.l_suppkey <> L1.l_suppkey
                   AND L3.l_receiptdate > L3.l_commitdate )
GROUP BY s_name
ORDER BY numwait DESC, s_name
```

The plan selected by ASE in Default mode has a hash join algorithm to join the tables orders and lineitem(L2), if all the join algorithms viz. nested-loop join, merge join, hash join are enabled(dss mode).

WCC model over iterations identifies that index orders_x is mostly cached, and therefore it is better to do a nested-loop join on orders_x rather than hash join. This will also help in preserving the needed ordering for the grouping done in operators above this join node. It can be seen from Figure 7 that the new plan gives about 11% advantage to that of the default plan.

To ensure that the plan selected by WCC is actually better, we ran the experiment by enabling only nested-loop join and merge join (mix mode).

when the same query is ran in mix mode, we identified that around 20 iterations, almost 75 percent of orders_x is being cached. Therefore, it is better to use lineitem(L2) before orders in the join order and use index orders_x to access the orders table so that the index lookup using orders_x becomes cheaper though there are more rows to join.

**TPCH Query 4:**

```sql
SELECT o_orderpriority, count(*) as order_count
FROM orders
WHERE o_orderdate >= '1995-04-01' 
  AND o_orderdate < dateadd(month, 3, '1995-04-01')
  AND EXISTS (SELECT *
               FROM lineitem
               WHERE l_orderkey = orders.o_orderkey
               AND l_commitdate < l_receiptdate )
GROUP BY o_orderpriority
```

Figure 6: Query 16 execution time

Figure 7: Query 21 execution time - dss mode

Figure 8: Query 21 execution time - mix mode
To verify the behavior of PIOC values when an object moves out of cache, after it was learnt that it is hot, we hand-tuned PIOC values for index lineitem_x and table LINEITEM to 0.5 to imply that 50% of the index and the data are cached.

When we re-ran the benchmarks, initially WCC model used to select lineitem_x as part of the best plan. This plan is same as that of the plan selected in No PIO mode, which assumes that the whole database is available in cache and no PIO is spent for accessing any object. Over iterations, WCC model identified that the actual-PIO spent on lineitem_x is greater then estimated. Also, since index lineitem_orders is being used to access LINEITEM table in many other queries, PIOC values for this index moved towards 0 more quickly. This allowed the cost of index lineitem_orders to be cheaper than that of lineitem_x. Therefore, we can see from Figure 9 that WCC model after about 100 iterations started selecting lineitem_orders as part of its best plan.

In general, WCC model over iterations favors objects constantly found in the cache, if the cost advantages because of PIO saving is better than the cost of other competitive plans.

On an average, we have observed that over iterations, the overall time for each stream has reduced in WCC mode. With-in first few iterations, mostly because of the influence of change in plan in query 16 discussed above, the average stream time has come down. From the Figure 10, we can see that after about 100 iterations, because of plan change in many other queries, the stream time has come down still further.

6. EXTENSIONS

This section describes some of the possible extensions that can be implemented over the proposed techniques.

PIOC granularity: It can be noted that though WCC model maintains PIOC values at a partition level, it is possible for a very selective point query with in a single partition to move the PIOC values for that partition from 1 to 0, there by suggesting that the data of the whole partition is cached. This problem arises, because the buffer manager operates at a page-level, while the PIOC values operate at a partition level. It could be addressed by allowing PIOC values to be learnt only on those queries which scan at least some threshold percentage x% (probably 5%) of the partition. It can be noted that it is not necessary for the query to have selectivity to be greater than this x%, but it is sufficient if the scan at least visits x% of pages of the partition. We would like to explore other possibilities of addressing this granularity gap.

Plan Caching: During our experiments, we have identified that there are ample of queries, for which, there is a single plan dominating other candidate plans across the range of PIOC values, i.e., that plan is optimal under most circumstances with PIOC values ranging from the object being partially cached to being completely cached. For example, it can be noted from Figure 6 that selecting a sequential scan of the table in part is a better choice for evaluating that query for PIOC values ranging from 0.9 – 0. Therefore, it is probably better to cache that plan as the candidate plan for that query, irrespective of the cache contents.

PIOC Freezing: Generally in production database systems, there would be a defined workload pattern that is observed. In those systems, it might be possible that there are few set of tables(fractions of tables) that are always hot, indicating that there is only a small portion of the cache, which is dynamic and constantly changing. In such scenarios, it might be better to just identify(learn) PIOC values for those hot tables and once the performance of the system is satisfactory with respect the selection of plans, stop the learning newPIOC by freezing the PIOC values. Later these PIOC values could be used as representative of cache occupancy for future optimizations. This might be useful when PIOC values are not changing much, since the cache has reached a steady state and learning minor swings in cache occupancy(PIOC) is not beneficial.

7. CONCLUSIONS

Growth of main memories have contributed to the possibility of a large fraction of databases being available in the buffer pool. We referred to previous work in this area, which define analytical models, to demonstrate the possibility of optimizer picking wrong plans.

We argued that though previous models which are proposed for determining buffer pool contents give fairly exact predictions, the cost
of obtaining such information is high. We also articulated that the buffer caches are modeled in such a way that most frequently accessed objects are always available in the cache and an approximate estimate of cache occupancy is good enough measure, especially if such information can be obtained with least possible cost.

In that direction, we defined WCC model as a novel feedback optimization technique for determining buffer cache occupancy and use that information in optimizing future queries. We have experimentally demonstrated the advantages of this technique, using TPC-H query suite. We have also depicted that the overhead because of our techniques is negligible when compared to the advantages it provides with respect to the execution time of the queries.

8. REFERENCES