A Query Beehive Algorithm for Data Warehouse Buffer Management and Query Scheduling

Amira Kerkad
kerkada@ensma.fr
Ladjel Bellatreche
bellatreche@ensma.fr
Pascal Richard
richardp@ensma.fr
Carlos Ordonez
ordonez@cs.uh.edu
Dominique Geniet
dgeniet@ensma.fr
LIAS/ENSMA - University of Poitiers, Futuroscope 86960, France
University of Houston, Houston, TX 77204, U.S.A.

ABSTRACT

Analytical queries, like those used in data warehouses and OLAP, are generally interdependent. This is due to the fact that the database is usually modeled with a denormalized star schema or its variants, where most queries pass through a large central fact table. Such interaction has been largely exploited in query optimization techniques such as materialized views. Nevertheless, such approaches usually ignore buffer management and assume queries have a fixed order and are known in advance. We believe such assumptions are too strong and thus they need to be revisited and simplified. In this paper, we study the combination of two problems: buffer management and query scheduling, in both static and dynamic scenarios. We present an NP-hardness study of the joint problem, highlighting its complexity. We then introduce a new and highly efficient algorithm inspired by a beehive. We conduct an extensive experimental evaluation on a real DBMS showing the superiority of our algorithm compared to previous ones as well as its excellent scalability.

Keywords: Data warehousing, query optimization, buffer management, query scheduling, query interaction.

INTRODUCTION

Database systems are facing an important growth of data that brings the need of high performance to a center stage. This growth is associated to an increasing number of complex analytical queries. Queries in such workloads may have common sub-expressions that are shared across multiple queries, which may have a significant impact on performance when they are reused. Data warehouses are a typical environment for large scale data and query interaction. This is generally due to the nature of their schema, usually a star, where no direct link exists between dimension tables. This feature makes all joins be based on the fact
Table, increasing the probability of finding common sub-expressions among different query plans. Query optimizers generally consider queries in an isolated way (one at a time) during the optimization process, which misses identifying common operations, resulting in redundant computations (Thomas, Diwan & Sudarshan, 2006). Multiple Query Optimization (MQO) tackled in (Sellis, 1988) brings a new dimension for database optimization by highlighting the interaction between queries. MQO has been exploited in several well-known problems such as:

- Materialized Views Selection Problem (MVSP) (Yang, Karlapalem & Li, 1997), that aims at identifying common sub-expressions to be materialized on disk so that they are not computed repeatedly.
- Buffer Management Problem (BMP) (Cornell & Yu, 1989), that defines the allocation and the replacement policy to manage retrieved data in a limited buffer space so that it can be reused as much as possible by other queries before it is evicted from the buffer pool.
- Query Scheduling Problem (QSP) (Gupta, Sudarshan & Viswanathan, 2001), that defines an order to evaluate a set of queries to take benefit from current content of a storage device before relevant data is evicted. The device may be a main memory buffer, or a secondary memory device such as hard disk, cloud, flash etc. For instance, when the storage device is a buffer, the QSP interacts with the BMP. In contrast, when the device is a disk, the QSP interacts with the MVSP (see Figure 1).

In (Scheuermann, Shim & Vingralek, 1996), the authors argue that in a data warehouse, caching entire sets of retrieved queries is more efficient than individual pages. One of the requirements to exploit query interaction is to identify overlapping parts between query plans. Once commonalities are identified, new challenges are faced to manage them using either one (isolated selection) or many (multiple selection) optimization techniques. In the former, we can find MVSP (Yang, Karlapalem & Li, 1997), BMP (Cornell & Yu, 1989), and QSP (Gupta, Sudarshan & Viswanathan, 2001). The latter combines several techniques such as MVSP with QSP (Phan. & Li, 2008), and BMP with QSP (namely BMSQP) (Thomas, Diwan, & Sudarshan, 2006). The large search space makes each problem hard to solve either in an isolated way or combined with another. But on the other hand, the interdependency between some techniques allows to bring performance beyond. For instance, the BMP is identified as highly interdependent with the QSP. Indeed, the buffer content determines the next query to run, and the schedule determines the buffer content. BMP and QSP have been studied in several contexts of databases: traditional (Effelsberg & Härder, 1984), data warehouses (Scheuermann, Shim & Vingralek, 1996), semantic databases (Yang & Wu, 2011), and flash databases (Ou, Härder & Jin, 2010) etc. Recently, the two problems have been combined (Thomas, Diwan, & Sudarshan, 2006). These techniques have been considered at three levels:

- Off-line: where the workload is known beforehand. This case allows defining some efficient buffer management policies because the utility of sub-expressions is given.
- Dynamic: where the system is adapted considering the new queries and data. It depends on analyzing the system to predict the utility of sub-expressions. For example, the cache hit ratio can be adjusted depending on the new workload.
- On-line: where the optimization process is run during the system execution to improve the performance.
of incoming queries on demand. At this level, the workload may either be known or not. In the latter case, the decision problem is much harder and needs advanced buffer management policies. Contrary to the two first levels, the on-line optimization is not widely studied in both BMP and QSP. Moreover, to the best of our knowledge, no work has tackled the combined problem: BMQSP.

**Contribution and Outline of the Paper**

In this paper, we study the BMP and QSP to propose a new approach for the simultaneous solution of these problems off-line and on-line. We prove that our problem is NP-complete and introduce a new algorithm inspired by a bee colony behavior, called Queen-Bee. This algorithm is based on a divide and conquer strategy.

The approach is proposed at two levels: first, we define our Queen-Bee algorithm in the off-line BMQSP. Then, we extend our proposal to work with the on-line case. The evaluation of our proposal efficiency and effectiveness is done on a large scale database twice: (i) first, theoretically, with a cost model and, and (ii) deploying the simulation results on a real DBMS with a specific script.

The rest of this paper is organized as follows: In Section 2, we discuss related work. Then, a background section is given containing the formal definition of the problem, hardness study and cost model details. Section 4 deals with the BMQSP off-line and on-line, where motivation examples are given to facilitate the understanding of our proposal; followed by the proposed algorithms. Section 6 presents intensive experiments using a simulation tool and a real DBMS. Section 6 concludes the paper by summarizing the main results and suggesting future work.

**RELATED WORK**

In recent years, the query interaction has been studied in several works (Ahmad et al. 2011; Phan & Li, 2008; Wangchao Le et al. 2012; Yang, Karlapalem & Li, 1997). This interaction is common in Multiple Query Optimization (MQO) problem and physical design in RDW (Sellis, 1988). The MQO is related to other optimization problems such as Buffer Management (BMP) (Cornell & Yu, 1989) and Query Scheduling (QSP) (Chipara, Lu & Roman, 2007; Märtens, Rahm & Stöhr, 2002), and the joint problem of BMP and QSP (Gupta, Sudarshan & Viswanathan, 2001; Tan & Lu, 1995; Gupta, Sudarshan & Viswanathan, 2001). These problems are studied in three main levels: off-line, dynamic (adaptive) and on-line. We present the related work of each level.

**Off-line Optimization**

In this level, the workload is supposed to be known beforehand. The isolated and combined problems have been largely studied in this category. The BMP has been studied in traditional databases (Sacco & Schkolnick, 1986; Effelsberg & Härder, 1984; Chou & DeWitt, 1985; Cornell & Yu, 1989), semantic databases (Yang & Wu, 2011), data warehouses (Scheuermann, Shim & Vingralek, 1996) and flash databases (Ou, Härder & Jin, 2010). Earlier works considered buffer management using some operating systems policies such as LRU (Least Recently Used), which is not suitable for database systems (Garcia-Molina, Ullman & Widom, 2008; Scheuermann, Shim & Vingralek, 1996) because it does not consider MQO. In (Scheuermann, Shim & Vingralek, 1996), a new buffer management approach called Watchman is proposed. The authors argue that in data warehouse environment, caching entire sets of retrieved queries is more interesting than individual pages. The proposed approach achieves better performance compared with LRU policy. In the second generation, some research efforts were concentrated on incorporating MQO in the buffer allocation (Cornell & Yu, 1989), where algorithms for selecting common intermediate results that need to be cached in a limited cache space were proposed.

The QSP has also been studied in several environments: centralized (Thomas, Diwan & Sudarshan, 2006), distributed and parallel databases (Märtens, Rahm & Stöhr, 2002). It has been proved as strongly NP-complete problem (Thomas, Diwan & Sudarshan, 2006). The interdependency between buffer management and query scheduling has been...
detected and studied in (Gupta, Sudarshan & Viswanathan, 2001; Tan & Lu, 1995) in the context of RDW and proposed some heuristics. (Gupta, Sudarshan & Viswanathan, 2001) presented several issues related to the combination of BMP and QSP by considering MQO problem. A complete hardness study was established and some algorithms without a real validation are proposed.

**Dynamic Optimization**

In this level, efforts are done adapting the system (off-line or on-line) considering the new data and queries. For example, the cache hit ratio is adapted dynamically when the workload changes. The dynamic optimization aspects have been considered in either buffer tuning without QSP (Faloutsos, Ng & Sellis, 1995; Schnaitter et al., 2006; Storm & Garcia-arellano, 2006; Suh, Rudolph, & Devadas, 2002), or in QSP disregarding the buffer (Phan. & Li, 2008).

In (Schnaitter et al., 2006), a framework called COLT is proposed for adjusting system configuration depending on the workloads and using some heuristics. The framework modifies existing optimization structures, such as indexes, by getting performance statistics. The query interaction and scheduling are not considered, although it may increase the performance of existing optimization structures. Other works such as (Faloutsos, Ng & Sellis, 1995; Storm & Garcia-arellano, 2006; Suh, Rudolph, & Devadas, 2002) propose adaptive buffer tuning techniques considering the arrival of new workloads and new statistics in order to improve buffer hit ratio.

Dynamic query scheduling got less attention compared to off-line query scheduling. In (Phan. & Li, 2008), the materialized views selection is mixed with query scheduling problem. The approach consists in scheduling queries and materializing/dematerializing views on-demand in order to satisfy the workload requirements.

**On-line Optimization**

In this level, the optimization process runs during the system execution. Few works studied the on-line QSP (Macdonald, Tonellotto, & Ounis, 2012) and the on-line BMP (Tran et al., 2008) in isolated way. Existing works consider resource allocation without query interaction. (Macdonald, Tonellotto, & Ounis, 2012), a framework is proposed to predict query efficiency using linear regression, and to schedule queries across replicated servers in order to minimize the average query waiting time.

Previous works on on-line BMP mainly deal with buffer tuning using: simulation, black-box control, gradient descent or empirical equations. Contrary to traditional methods, a new analytical-derived equation has been proposed in (Tran et al., 2008) to allocate the buffer in such way to minimize miss probability $p_{miss}$. The equation is used for dynamic self-tuning to adapt new queries faster. The equation allows also identifying the buffer size limit (upper bound) for buffer partitioning, ensuring fairness in buffer reclamation and dynamically retuning the allocation when workloads change. This work considers the BMP in an isolated way ignoring the impact of query order and interaction on performance. In addition, the prior knowledge of workloads allows defining better BMP policies rather than the commonly used in operating systems, which are not suitable for database systems (Garcia-Molina, Ullman & Widom, 2008).

To the best of our knowledge, no work has considered the joint problem on-line. Table 1 summarizes our discussion on related work, and shows that only few approaches address the on-line case.

<table>
<thead>
<tr>
<th>Off-line</th>
<th>BMP</th>
<th>QSP</th>
<th>BMQSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sacco &amp; Schkolnick, 1986;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schuermann, Shim &amp; Vingralek, 1996;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yang &amp; Wu, 2011.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 1. Classification of existing works on BMP and QSP

Based on this study, we propose to deal with the static case (off-line BMQS) in order to propose a new algorithm that may be adapted to solve the on-line BMQS problem.

BACKGROUND

Multiple View Processing Plan

Each query may be represented by an algebraic tree corresponding to its execution plan. Due to the strong interaction between queries, their plans may be merged to generate a graph structure called, Multiple View Processing Plan (MVPP) (Yang, Karlapalem & Li, 1997) (Figure 2). The leaf nodes of the MVPP represent the tables of the RDW. The root nodes represent the final query results and the intermediate nodes represent the common subexpressions shared by the queries. Among intermediate nodes, we distinguish:

(i) unary nodes representing the selection (σ) and the projection operations

(ii) binary nodes representing join (⋈), union, intersection, etc.

Figure 2: Multiple View Processing Plan for a workload of 10 queries.

This representation allows mainly to show up the overlapping subexpressions between queries. Note that, in our study we consider the star join queries running on a star schema data warehouse. In such schema, all joins pass by the fact table. In Figure 2, a workload of 10 star join queries is running on a star schema with a fact table $F$, and three dimension tables: $D1$, $D2$ and $D3$. We notice that the join operation in node $n_i$ is shared between $Q1, Q4, Q6$ and $Q7$. This node can be computed once and cached to contribute in lowering the execution cost of all queries accessing it.

Formalization

Before considering the joint problem, and to facilitate the understanding of its formalization, we start presenting a separate formalization of both BMP and QSP.

BMP is formalized as follows:

- Inputs: (i) RDW, (ii) a workload with a set of queries $Q = \{Q_1, Q_2, \ldots, Q_n\}$ represented by a MVPP, (iii) a set of intermediate nodes of MVPP candidates for caching $N = \{n_0, n_1, \ldots, n_i\}$;
- Constraint: a buffer size $B$;
- Output: a buffer management strategy $BM$ that allocates nodes into the buffer to optimize the cost of processing $Q$.

QSP is formalized as follows:
- Inputs: (i) RDW, (ii) a workload with a set of queries \(Q = \{Q_1, Q_2, ..., Q_n\}\) and (iii) a device management policy (the device can be either a buffer or disk);

- Output: scheduled queries of the workload \(Q\) into a new ordered set: 
  \(SQ = \{SQ_1, SQ_2, ..., SQ_n\}\), having the least execution cost.

Given a workload of \(n\) queries, we propose to formalize the joint problem on two levels: off-line and on-line. Off-line BMQSP is described using the above formalizations as follows: The joint problem takes (i) A RDW and (ii) a set of queries \(Q = \{Q_1, Q_2, ..., Q_n\}\) represented by the MVPP, (iii) a set of intermediate nodes of MVPP candidates for caching \(N = \{n_0, n_1, n_2, ..., n_{\rho}\}\); The constraint is the limited buffer size \(B\). The problem aims at providing: (i) a scheduled set of queries \(SQ = \{SQ_1, SQ_2, ..., SQ_n\}\) and (ii) a buffer management strategy \(BM\), minimizing the overall processing cost of \(Q\). On-line BMQSP is formalized in the same way, but considering the on-line aspect while executing queries: The joint problem takes (i) A RDW and (ii) an arriving set of on-line queries \(Q = \{Q_1, Q_2, ..., Q_n\}\) represented by the MVPP, (iii) a set of intermediate nodes of the MVPP candidates for caching \(N = \{n_0, n_1, n_2, ..., n_{\rho}\}\); The constraint is the current available buffer size \(B\). The problem aims at providing: (i) on-line scheduled queries \(SQ = \{SQ_1, SQ_2, ..., SQ_n\}\) and (ii) a buffer management strategy \(BM\), minimizing the overall processing cost of \(Q\) on-line.

**Hardness study**

In this section, we prove that the decision problem of BMP is NP-complete in the strong sense, meaning that neither polynomial time nor pseudo-polynomial time algorithms can be defined for solving it, unless \(\mathcal{P} = \mathcal{NP}\). The hardness proof is based on a transformation from an interval packing problem (i.e., INTVPACK-WEIGHT) known to be NP-Complete in the strong sense. In this interval problem, the objective is to select the maximum number of weighted intervals, subject to the constraint that for any point in time, the total weight of the chosen intervals does not exceed a given threshold (Chrobak et al., 2012). That problem has been used for establishing the hardness for several multi-size page caching problems. We now give a formal definition of this pivot problem.

We consider \(N\) intervals \((s_i, t_i)\), denoted by their indices \(1 \leq i \leq N\) with a weight \(w_i\). For a subset \(S \subseteq \{1, ..., N\}\) of intervals, the weight of \(S\) is \(w(S) = \sum_{i \in S} w_i\) and the cut at time \(\gamma\) is the set of intervals that contains \(\gamma: \text{cut}_\gamma(S) = \{i : s_i < \gamma < t_i\}\). The weighted interval packing problem is to choose the maximum-weighted subset of intervals satisfying \(w(\text{cut}_\gamma(S)) \leq W\).

**Problem:**

- INTVPACK-WEIGHT (Chrobak et al., 2012).

**Instance:**

- A set of \(N\) open intervals \((s_i, t_i)\) for \(i = 0, ..., N - 1\), where each interval \(i\) has weight \(w_i \geq 0\). Positive integers \(W\) and \(L\).

**Question:**

- Is there a subset \(S\) of the intervals with \(w(S) \geq L\) that satisfies \(w(\text{cut}_\gamma(S)) \leq W\) for any real number \(\gamma\)?

The INTVPACK-WEIGHT has been shown NP-Complete in the strong sense using a transformation from the well-known VERTEX COVER problem (Garey & Johnson, 1979).

**Theorem 1:** (Chrobak et al., 2012) **INTVPACK-WEIGHT is NP-Complete in the strong sense.**

We now define the NP-Completeness result for the decision problem of Buffer Management (BMP). The reduction mainly follows the principles used in (Chrobak et al. 2012) for proving the NP-Completeness of a multi-size page caching problem.
We first define the decision problem corresponding to the Buffer Management problem. For sake of clarity, we assume (in the hardness study only) a basic cost model in which the cost of loading an uncached node is equal to its size.

**Instance:**
- A set of $N$ open intervals $(s_i, t_i)$ for $i = 0, ..., N - 1$, where each interval $i$ has weight $w_i \geq 0$. Positive integers $W$ and $L$.

**Problem:**
- **BMP:** Buffer Management Problem;

**Instance:**
- A set of queries $\{Q_1, Q_2, ..., Q_m\}$ with a set $N = \{r_1, r_2, ..., r_l\}$ of nodes in its execution plan with cost and size $\text{Cost}(n_i) = \text{Size}(n_i)$; $(1 \leq i \leq l)$, a sequence of operations for a given execution plan $\{r_1, r_2, ..., r_m\} \in N$, a buffer of size $B$ and a cost bound of $F$.

**Question:**
- Is there a replacement policy that serves $\{r_1, r_2, ..., r_m\}$ with a buffer size of $B$ and a cost of at most $F$?

**Theorem 2:** BMP is NP-Complete in the strong sense.

**Proof:** It is easy to see that BMP is in $\mathcal{NP}$, since a non-deterministic algorithm (i.e., a Non-Deterministic Turing Machine) needs only to guess the subset of nodes to be cached and then evaluates in polynomial time if the total cost is at most $F$. Let us construct an instance of BMP from a generic instance of INTVPACK-WEIGHT. For every interval number $i$, denoted $(s_i, t_i)$, we define a node $n_i$, $0 \leq i \leq N - 1$ with a size $\text{SIZE}(n_i) = w_i$. The operation sequence generated by queries consists in $2N$ requests $r_1, r_2, ..., r_{2N}$. Every node $n_i$ is requested twice: the first time at the instant $s_i$ and the second one at time $t_i$. The cache size is $B = W$ and the cost bound is

$$F = 2 \sum_{i=0}^{N-1} w_i - L \tag{1}$$

The proof obligation is: INTVPACK-WEIGHT has a solution if, and only if, BMP has a solution from the above construction.

**If:** Assume that the INTVPACK-WEIGHT has a solution $S$. We now prove that the corresponding solution is valid for BMP. During the operation sequence, only the node associated to $S$ will be loaded in the cache: $n_i$ is loaded at time $s_i$ and flushed at time $t_i$ after its second and last use. The condition that $w(\text{cut}_i(S)) \leq W$ for the solution $S$ of INTVPACK-WEIGHT ensures that loaded nodes never exceed the cache capacity $B = W$. The total cost is defined by: every node in $S$ is loaded once time in the cache leading to a cost of $w(S)$, whereas nodes that are not in $S$, denoted $\bar{S}$ are loaded twice in the memory outside the cache leading to a cost of $w(\bar{S}) \leq L$. Since $S$ is a solution of INTVPACK-WEIGHT, we have $w(S) \leq L$. Since $N = S \cup \bar{S}$, the total cost is bounded by:

$$F = w(S \cup \bar{S}) \leq 2w(N) - L = 2 \sum_{i=0}^{N-1} w_i - L \tag{2}$$

So, a valid BMP solution has been exhibited.

**Only If:** Assume that BMP has a solution. Then, we prove that a valid solution can be defined for the problem INTVPACK-WEIGHT. Every node is requested two times. Let $S$ be defined by the set of nodes that do not generate a fault when they are requested the second time. Such nodes are necessarily in the cache. Let $S$ defines a subset of the $N$ original intervals corresponding to these nodes. Since the total cost is bounded by:

$$F = 2 \sum_{i=0}^{N-1} w_i - L$$

This implies that $w(S) \geq L$ and as consequence the first constraint of INVPACK-WEIGHT is satisfied. Secondly, the nodes
\( i \in S \) are stored in the cache during the whole time in the open interval \((s_i, t_i)\) without generating a fault. Thus, the cache capacity \( B = W \), the weight of the cut at any time \( \gamma \) satisfies the constraint:

\[
\omega(\text{cut}_\gamma(S)) \leq W \quad (3)
\]

So, we have defined a valid solution for the decision problem INTVPACK-WEIGHT

**Theorem 3:** The combined problem of Buffer Management and Query Scheduling off-line and on-line are NP-Complete in the strong sense.

Note that, in (Gupta, Sudarshan & Viswanathan, 2001), a proof is given of NP-Completeness of the problem of Query Scheduling Problem (QSP).

**Cost Model**

To quantify the quality of the solutions, we define a cost model to estimate the number of inputs/outputs (I/O) required for executing the set of OLAP queries. We describe the cost model in three levels: (1) basic estimation, (2) considering the buffer and (3) considering on-line execution.

**a) Basic estimation**

Most studies using cost models for selecting optimization structures ignore buffer management and on-line aspects. Therefore, the cost of a given query \( Q_i \) may be estimated as the sum of the costs of all joins, aggregations and projections. More concretely, the cost of executing a query \( Q_i \) involving several joins between the fact table \( F \) and the dimension tables \( D^Q_i = \{D^Q_{i_1}, D^Q_{i_2}, \ldots, D^Q_{i_n}\} \) is composed of three types of operations: (1) first join \((FJ)\), (2) intermediate result \((IR)\) and (3) final operation \((AG)\) (aggregation, group by). More details about joins order and estimating result sizes are available in (Kerkad, Bellatreche & Geniet, 2012).

We assume that each query \( Q_i \) is represented by an ordered set of all the operations of its plan:

\[
op^Q_i = \{\text{op}^Q_{i_1}, \text{op}^Q_{i_2}, \ldots, \text{op}^Q_{i_l}\}
\]

\( \text{op}^Q_{i_1} \in \{FJ, IR, AG\} \) from the first join until the final operation, with \( l_i \) the number of operations in \( Q_i \). We define a function \( \text{size}(\text{op}^Q_{i_k}) \) estimating the result size of \( \text{op}^Q_{i_k} \). Recursively, we estimate the cost of a query starting from the final operation \( \text{op}^Q_{i_l} \) as follows:

\[
\text{Cost}(\text{op}^Q_{i_k}) = \begin{cases} 
\text{size}(\text{op}^Q_{i_1}) & \text{if } k = 1 \\
\text{size}(\text{op}^Q_{i_k}) + \text{Cost}(\text{op}^Q_{i_{k-1}}) & \text{if } k \in [2, l_i]
\end{cases}
\]

**b) Considering buffer content**

To take into account buffer management in our cost model, the cache content needs to be checked. The cost of an operation \( \text{op}^Q_{i_k} \) is ignored if: (i) Its result is cached, or (ii) one of its successors \( \text{op}^Q_{i_{k+\alpha}} \) with \( \alpha > 1 \) is cached. For this reason, we define a function \( b(\text{op}^Q_{i_k}) \) to check if the result of \( \text{op}^Q_{i_k} \) is present in the buffer (=1) or not (=0). Therefore, the execution cost will be:

\[
\text{Cost}(\text{op}^Q_{i_k}) = \begin{cases} 
\text{size}(\text{op}^Q_{i_1}) \times \prod_{j=k}^{l_i} b(\text{op}^Q_{j}) & \text{if } k = 1 \\
\text{size}(\text{op}^Q_{i_k}) + \text{Cost}(\text{op}^Q_{i_{k-1}}) \times \prod_{j=k}^{l_i} b(\text{op}^Q_{j}) & \text{if } k \in [2, l_i]
\end{cases}
\]
c) Considering on-line execution

When queries are executed concurrently, the arrival of a new set of queries triggers a scheduling process to reorder the queue and to decide which query to execute after the one that is being executed. This repeatedly processed scheduling may cause a delay increasing response time. This delay depends on the algorithm’s greediness. The required cost/time to schedule the incoming queries is added as an extra cost increasing completion time, noted \( \text{Delay}(Q') \) where \( Q' \subseteq Q \). A query listener captures the set of arriving queries \( AQ \). If \( AQ \) is empty, no scheduling is required and no extra cost is added (\( \text{Delay}(AQ) = 0 \)). Otherwise, the listener triggers the scheduler and the elapsed time is added in terms seconds to the completion time (\( \text{Delay}(AQ) > 0 \)) (the equivalent extra cost in terms of I/O is added to the total cost).

\[ \text{Total Cost} = \sum_{i=1}^{n} \text{Cost}(op_{i}^{Q}) + \text{extra cost}(AQ_{i}) \quad (6) \]

The corresponding completion time is obtained as the time of executing each query and the elapsed time for scheduling the new arriving ones:

\[ \text{Completion Time} = \sum_{i=1}^{n} \text{Time} (\text{Cost}(op_{i}^{Q})) + \text{Delay}(AQ_{i}) \quad (7) \]

**BUFFER MANAGEMENT AND QUERY SCHEDULING**

In this section, we present an integrated solution for managing the database buffer and scheduling queries: off-line and on-line.

**Off-Line Scenario**

The total execution cost of the workload is estimated as the sum of all queries costs and the extra cost required for scheduling.

Figure 3: Clustering queries by interaction to obtain hives

To illustrate this scenario, let us consider the following example.

**Example 1:**

We assume a workload with 10 OLAP queries represented by a MVPP. The intermediate results of the MVPP are candidates for bufferization. To show the interaction between BMP and QSP, we propose to reorganize the
initial structure of the MVPP by generating clusters of queries; each cluster contains queries having at least one node in common. Each cluster is called a hive. Figure 3 shows the results of clustering of our MVPP, where three hives are obtained. In each hive, a carefully chosen query is selected to be the Queen-Bee. Once elected all its common nodes are cached. Thus queries in the same hive will be ordered and will benefit from shared buffer content.

Throughout this section, we detail the technical issues announced in this example: (1) identification of intermediate nodes, (2) generation of hives, (3) buffer allocation, and (4) query scheduling.

1. Queen-Bee Algorithm

To reduce the complexity of the combined problem, we propose an approach inspired from the natural life of bees (Kerkad, Bellatreche & Geniet, 2012). The methodology of resolution is illustrated in Figure 4.

We associate the dynamic buffer management strategy DBM (Kerkad, Bellatreche & Geniet, 2012) to our queen-bee algorithm. DBM mainly traverses the query plan and, for each intermediate node (operation), it checks the buffer content: if the result of the current operation is already cached, then no need to load it from the hard disk, else it is cached while there is enough buffer space. Once a node of a given hive is treated, its rank value is decremented. When the rank of a node is equal 0, it will be removed from the buffer since it is useless for coming queries.

Our query scheduler works on the clusters of queries (hives) and it shall order the queries inside each hive according to the buffer content. To do so, three modules define our queen-bee algorithm: (1) generating a query graph with connected components (QGCC), (2) Inter-Component Scheduling (ICS) which is optional depending on whether queries have priority or not and (3) sorting queries inside each component by Local Optimizer (LO).

2. Generating the Query Graph with Connected Components

Starting from MVPP, a QGCC representing hives is obtained as follows:

- Vertices represent the queries of each hive.
- Each vertex is tagged with a value $C$ representing the I/O cost of its corresponding query.
- An edge exists between two vertices if they have common node(s). It is labeled by the number of the shared nodes.

Figure 5 shows the corresponding QGCC for the MVPP in Figure 2.

We choose to present our approach incrementally since it concerns two main problems: BMP and QSP. The basic idea behind our queen-bee algorithm is to partition the queries of the MVPP and then for each hive, elect a query (queen-bee) to be executed first and its nodes will be cached.

Definition 1: we define the rank value of a node $no_i$ as a counter representing the number of queries accessing $no_i$ in a period of time.

Figure 4: Methodology of resolution for off-line BMQS

Figure 5: An example of QGCC
3. Inter-Component Scheduling

In our study, all queries have the same priority. Therefore, the required data for two different hives are disjoint. For example:

\{(Q1,Q7,Q4,Q6); (Q3,Q9,Q10); (Q2,Q8,Q5)\}

\{(Q3,Q9,Q10); (Q1,Q7,Q4,Q6); (Q2,Q8,Q5)\}

Figure 6: Local Optimizer on a connected component using minimal cost criterion

4. Local Optimizer

The queries inside each component need to be scheduled. For this reason, LO takes each component and schedules its queries by performing three steps:

- Identification of the Queen-Bee based on the chosen criterion (minimal cost).
- Exploration of the rest of nodes (using the same criterion used for selecting the queen-bee) by considering the current content of the buffer.
- Once the connected component is entirely traversed, return the obtained sub-schedule.

Figure 6 gives an example of LO using the cost criterion in ascending order. When we traverse the component starting from the query having the minimal execution cost, the cost of vertices which are not traversed yet is updated depending on cache content. The next vertex to be visited is the one with minimal cost. The algorithm repeats the operation while the component is not entirely traversed. This sort "sacrifices" the query with minimal cost by executing it first. This is because it will not get any relevant data in cache. But, this same query will give relevant data for the other queries which are more expensive, because they share at least one overlapping node. In Figure 6, the total cost is:

\[1000 + 1000 + 1000 + 1800 = 4800 \text{ I/O}.\]

5. Queen-bee Complexity

The Queen-Bee Algorithm has three main phases. In the first phase, the QGCC is constructed for the queries. Let be the number of nodes in the MVPP of all the query plans. For each query, the existing nodes in its plan are explored and compared with the other \((n - 1)\) queries. The generation of the QGCC is performed in \(O(n^2 \times l)\).

The ICS phase is optional. The last phase is the Local Optimization. It takes each component and performs the following operations for its \(n_i\) queries:

**Exploration of the current component:** for each query the following operations are performed:

a) Check buffer content()
b) Update buffer content()
c) Get Cost()
d) Sort remaining queries by minimal cost
e) Choose next query

**Estimation of the execution cost:** this step requires exploring the execution plan for the current query, and for each node check the buffer content. The Cost estimation is performed in \(O(l)\).

**Sorting remaining queries** is done by Quick Sort in \(O(\log n_i)\).

The computation cost of other operations (a,b,e) is constant, thus the ICS for all components is done in \(O(n \times l + n \times \log n)\).
On-line Scenario

In order to keep the same presentation as for the off-line scenario, we consider a motivating example.

**Example 2:**

We extend the motivating example of the off-line optimization. Let’s consider the same workload in the on-line case, where queries are launched concurrently from different users. Several situations can be distinguished:

- **Case 1:** One query arrives in the system and the queue is empty. In this case the query is run and its intermediate results are cached in the available buffer space. No scheduling is needed.

- **Case 2:** \( N \) queries arrive and the queue is empty (\( N > 1 \)). An order is required to schedule concurrent queries based on their interaction and on the buffer content.

- **Case 3:** One or more queries arrive and other queries are already launched. The pending set of queries needs to be analyzed to insert the new queries into the schedule.

At the light of this example, we can extend the off-line scheduling properties to deal with the on-line case. In the example, the case 1 does not require any scheduling. In case 2, queries can be scheduled depending on the clusters (hives) to allow queries in the same hive to reuse overlapping subexpressions. In case 3, the existing hives, obtained from the already run queries, need to be adapted considering the new arriving queries. The new hives allow adjusting the schedule of pending queries by inserting the new ones into the schedule.

1. **On-line Queen-Bee**

To tackle the on-line BMQSP, we extend our Queen-Bee algorithm for off-line resolution in order to consider the on-line execution. Our methodology is represented in Figure 7.

**Buffer Management Policy:**

We associate a buffer management policy to our algorithms depending on the administrator knowledge.

1. If the administrator has no prior knowledge of the workload, then we use one of the most used policies in database systems, which is *Least Recently Used* policy (LRU).
2. Otherwise, if the administrator knows at least queries frequency, then we propose to use our dynamic buffer management (DBM) strategy in our algorithm (previously described in Off-line BMQS section).

![Figure 7: Our methodology for on-line BMQS.](image)

2. **Query Listener**

The query listener represents an intermediate module between the users and the waiting queue. It captures incoming user's requests before it lays in the queue. The MVPP is incrementally traced by merging the new query plans. This listener triggers the scheduling and buffering process on the set containing the already queued and the new arriving queries. We propose to evict double occurrence of queries from the queue. In the case where many users launch the same query, it is processed once because we consider only selection queries (no updates).

3. **Generating QGCC on-line**

Starting from obtained MVPP, two different cases are distinguished:

- If the QGCC is being constructed for the first time (initial set of queries), it is
generated in the same way as in the off-line case.

- If the QGCC already exists, it is updated by extending hives with new vertices. To identify the hive to whom belongs a new query, the algorithm checks its first join. A query may share at least the first join with the queries in the same hive. New hives can be added if some queries does not belong to any existing hive.

4. Inter-Component Scheduling

As described in previous study on off-line BMQS, this phase is applied only when queries have priority. Note that, if the administrator chooses to limit the delay for some queries, a constraint can be added to allow these queries to be executed before the delay time is exceeded. We do not consider this constraint in our study to simplify the understanding of our approach.

![Diagram of Inter-Component Scheduling](image)

Figure 8: Examples of running the the on-line queen-bee

5. Local optimizer

The queries inside each component need to be scheduled. For this reason, the Local Optimizer takes each updated hive and schedules its queries by performing three steps that are the same as in off-line BMQS. Example of running on-line Queen-Bee: Figure 8 illustrates some examples of running the on-line queen-bee algorithm (Algorithm 1) depending on the arriving queries and the queue content.

In case (a), $Q_1$ is added to the empty QGCC and no scheduling is required. In case (b), four queries arrive and no queries are left in the queue. The QGCC is updated and the set of new queries $AQ$ is scheduled. In cases (c) and (d), the new arriving queries update the QGCC and the scheduling processes them with queued queries to adjust the existing schedule.

EXPERIMENTAL EVALUATION ON A REAL DBMS

To evaluate our proposal, we conduct experiments on a simulation tool that uses our mathematical cost model and a real DBMS (Oracle11g) for both off-line and on-line scenarios. First of all, we present the dataset and the environments of our experimentations and then we detail each scenario.
Algorithm 1: On-line Queen-Bee Algorithm

1: Queue ← QueryListener();
2: If (QGCC is empty) then
3: \[ Q_{\text{wait}} = \text{Queue}; \]
4: \{create QGCC for the new queries\}
5: QGCC(Q_{\text{wait}});
6: \{define priorities between hives\}
7: ICS(Q_{\text{wait}}, \emptyset);
8: \{sort queries inside hives\}
9: LO(Q_{\text{wait}}, \emptyset);
10: Else
11: \[ Q_{\text{run}} = \text{previous}_\text{queries}; \]
12: \[ Q_{\text{wait}} = \text{Queue}; \]
13: \{update QGCC with new pending queries\}
14: Update_QGCC(Q_{\text{wait}}, Q_{\text{run}});
15: \{update priorities between hives\}
16: ICS(Q_{\text{wait}}, Q_{\text{run}});
17: \{scheduling inside modified hives\}
18: LO(Q_{\text{wait}}, Q_{\text{run}});
19: End if
20: Add_delay(); \{add elapsed time of scheduling\}
21: run(Q_{\text{run}});
22: get_cost(Q_{\text{run}});
23: refresh_queue();

Simulation tool

To make easier our testing, we developed a simulation tool using a Java development environment. It contains three main modules:

- RDW connectivity and meta-data extraction needed for the cost model,
- Setting RDW parameters and handle the workload,
- Optimization module handles different parameters (e.g. Buffer size), the choice of algorithms and the output detailing the obtained solutions for administrators. If they are not satisfied by these results, the output detailing the obtained solutions for administrators. If they are not satisfied by these results, the optimization module gives them the possibility to tune some parameters. Otherwise, administrators deploy them on Oracle11G using appropriate scripts.

1. Off-line Scenario: Obtained results

In (Kerkad, Bellatreche & Geniet, 2012), a hill climbing and genetic algorithms are proposed to get a near optimal query scheduling and its buffer management using respectively dynamic query scheduler (DQS) and dynamic buffer management (DBM).

Dataset and workload

Our experiments are done on the Star Schema Benchmark (SSB of 100GB) having a fact table Lineorder \((6,000,000 \times \text{SF tuples})\) and 4 dimension tables with a scale factor \(\text{SF} = 100\). A server with 32GB of RAM is used. We consider thirty queries, covering different types of read-only OLAP queries (star join queries with: 1, 2, 3 and 4 joins).

Figure 9: Comparing different algorithm’s performance
In Figure 9, our previous algorithms and the queen-bee are compared with results obtained using LRU policy instead of DBM, and no scheduling instead of DQS. Different buffer pool sizes are used to show its impact on the total performance. From these results, we observe the following:

- DBM is more adapted than LRU in our context;
- DQS evolves the efficiency of the buffer management policy;
- Queen-Bee gives the same performance as the greedy heuristic (genetic) that uses DQS and DBM.

We also notice that beyond a threshold of buffer pool space, query scheduling has no effect on the final performance because all candidate nodes can fit in the cache. That’s why the DBM without scheduling gives the same performance as DBM-DQS (genetic algorithm) and Queen-Bee at a buffer space of 20GB.

One of the motivations of our proposal is to prune the search space. In Figure 10, we can see that the queen-bee algorithm is much faster than the genetic algorithm using DBM even though they give the same query performance.

### Validation on Oracle11g

For validation, we deploy our simulation results on an Oracle11g DBMS with the same data set (SSB of 100GB and 30 queries). Queries are executed for each solution schema obtained by our algorithms. To perform this validation, queries are rewritten to take into account caching solution proposed by different algorithms. The DBMS parameters are tuned to prepare buffer pool. To illustrate the rewriting process, let us consider the following example:

#### Example 3:

Let Q20 be an OLAP query defined as:

```sql
--Q20:
select c_city, s_city, count(*) as revenue
from customer, lineorder, supplier
where lo_custkey = c_custkey and lo_suppkey = s_suppkey
and c_nation = 'UNITED STATES' and s_nation = 'UNITED STATES'
group by c_city, s_city
order by revenue desc;
```

If one of our algorithms identifies the node representing the join between Lineorder and Supplier; Q20 is then rewritten as follows using Hints of Oracle11g DBMS:

```sql
--Q20:
select c_city, s_city, count(*) as revenue
from(
  select /*+result_cache*/ * from lineorder, supplier
  where lo_suppkey = s_suppkey and s_nation = 'UNITED STATES'
) join customer on lo_custkey = c_custkey
where c_nation = 'UNITED STATES'
group by c_city, s_city
order by revenue desc;
```

Once a query is rewritten, the buffer needs to be allocated by the relevant nodes.
corresponding to that query before evaluating it using Oracle query optimizer. To do so, several SQL commands are required:

- buffer setting:
  
  ```sql
  alter system set db_cache size = 800 M;
  ```

- buffer flushing to remove all objects:
  
  ```sql
  execute dbms_result_cache.flush;
  ```

- caching node:
  
  ```sql
  ...select/*+result cache*/ * from ...
  ```

- node remove from the buffer:
  
  ```sql
  exec dbms_result_cache.invalidate object(3);
  ```

To compare LRU with the DBM, we take the same workload in a static order on Oracle11g with 6GB of buffer pool. The queries are executed using: (1) no buffer management, (2) LRU, and (3) our DBM. In Figure 12, we can observe that LRU policy gives a good performance for some queries but not enough to cover a larger number of queries as the DBM.

![Figure 12: LRU vs. DBM: experiments with static schedule](image1.png)

![Figure 13: Deploying Queen-Bee results](image2.png)
The validation of the simulation experience in Figure 12 is done on our DBMS. Real performance is given in Figure 11 which shows the similarity with the theoretical results. This proves the quality of our cost model.

Figure 13 shows the impact of Queen-Bee optimization on the workload performance. We can see that the number of queries that have no benefit from cache content is 6 of 30 queries. This corresponds exactly to the number of query hives of our workload. We notice that only the first query (the queen-bee) has no reduction in cost, but all the remaining queries have important performance gain because of sharing at least one common result.

2. On-line Scenario: Obtained results

![Figure 14: Running concurrent subsets of queries on-line](image)

![Figure 15: Total execution cost obtained by each optimization technique](image)

In Figure 14, several queries are launched concurrently in four timeframes. We note $T_i$ the event of getting new queries in a timeframe $t_i$. Thus, $T_1, T_2, T_3$ and $T_4$ represent the four new sets of queries that arrive during system life time. The execution of the arriving queries is processed in 4 baselines:

- No optimization case, where queries in the queue are processed without scheduling and buffering.
- LRU policy for managing the buffer without scheduling.
- on-line Queen-Bee using LRU policy, where the user have no prior knowledge of the workload.
- on-line Queen-Bee using Dynamic Buffer Manager (DBM), where the user knows the execution frequency of queries.

The results show that the new queries obtained after each new arrival $T_i$ are processed in a different way. The variation of the schedule or the buffer strategy has a significant impact on performance.

The total execution cost (Figure 15) shows the impact of buffering on performance. It also shows the interdependency between scheduling and buffering, where the buffer management policy performance is enhanced by scheduling. The Dynamic Buffer Management (DBM) policy is more efficient then LRU, but needs a prior knowledge of queries’ frequency. The results show that the DBM gives a better yield to the on-line-Queen-Bee algorithm then LRU.

Validation on a real DBMS

In Figure 16, the elapsed time of processing the new queries is given in milliseconds. The process takes into account the new set of queries, the buffer content and the already queued queries which are not run yet. The results show that the algorithm is very fast and the processing time is negligible compared to completion time for running queries (Figure 17).
In Figure 18, the obtained schedule using our algorithm (OQB-DBM) is run over a real database of the Star Schema Benchmark (100GB) located on a server of 32GB of RAM using Oracle11g DBMS. The buffer pool size is set to 20GB as in simulation experiments. The execution of queries shows that a subset of queries does not have any benefit from the optimization process. These queries are either (1) without any interaction with the others or (2) have no relevant object in the buffer (cache hit) or (3) are chosen as a queen-bee to satisfy other queries. Other queries have a significant reduction in their execution cost using the previously cached objects by queen-bees, which are kept in the buffer as long as possible by the DBM policy.

CONCLUSION & PERSPECTIVES
In the advanced applications build around databases like those used in data warehouses, scientific, statistic and OLAP, analytical queries are generally interdependent. Such interaction has been largely exploited in selecting optimization techniques such as materialized views. Most of the studies use cost models ignoring the buffer management and assume the pre-ordered know queries. In this paper, we study the off-line and on-line problem combining buffer management and query scheduling. Its hardness study is well presented. To deal with this complex problem, we propose a new and efficient
algorithm, called the Queen-Bee algorithm, inspired from bees behavior. The principle of this algorithm is used in off-line and on-line scenarios. In order to evaluate solutions, an advanced mathematical cost model is presented to allow predicting query execution cost considering buffer content, query order and on-line aspects. Our findings are integrated in a java simulation tool. The obtained results are deployed on a real database system (Oracle11g) running on a server with 32GB of RAM with the same data set used by our simulator. Two main interesting results can be highlighted: (i) the quality of our cost model since the theoretical performance coincides with the real ones and (ii) the high positive impact of considering the query interaction in managing the buffer and scheduling the queries in off-line and on-line scenario.

Currently, we are working on several issues that could have a profound impact on database performance: (i) the deployment of our proposal on another commercial DBMS (SQL Server) and on non-commercial DBMS like PostgreSQL, (ii) the incorporation of our findings in the process of selecting optimization techniques such as indexes and data partitioning (Bellatreche et al. 2009) and (iii) the consideration of update queries.

REFERENCES


Storm A. J. and Garcia-arellano C., (2006). Adaptive...
self-tuning memory in db2. In In Proc. of the 2006 Intl. Conf. on Very Large Data Bases.


