Queen-Bee: Query Interaction-Aware for Buffer Allocation and Scheduling Problem

Amira Kerkad, Ladjel Bellatreche, and Dominique Geniet

LIAS/ENSMA Poitiers University, Futuroscope, France
{amira.kerkad,bellatreche,dgeniet}@ensma.fr

Abstract. In the relational data warehouses, each OLAP query shall involve the fact table. This situation increases the interaction between queries that can have a significant impact on the warehouse performance. This interaction has been largely exploited in solving isolated problems like (i) the multiple-query optimization, (ii) the materialized view selection, (iii) the buffer management, (iv) the query scheduling, etc. Recently, some research efforts studied the impact of the query interaction on optimization problems combining interdependent sub-problems such as buffer management problem (BMP) and the query scheduling problem (QSP). Note that combining two complex problems usually increases the complexity of the integrated problem. In this paper, we study the effect of considering the query interaction on an integrated problem including BMP and QSP (namely, BMQSP). We first present a formalization of the BMQSP and show its hardness study. Due to high complexity of the BMQSP, we propose an algorithm called queen-bee inspired from the natural life of bees. Finally, theoretical and effective (on Oracle 11G) experiments are done using the star schema benchmark data set.

Keywords: Multi-query optimization, Query Scheduling, Buffer Management, Query Interaction.

1 Introduction

Relational data warehouses ($\mathcal{RDW}$) represent the ideal environment in which complex OLAP queries interact with each other. These queries often have a lot of common sub-expressions, either within a single query, or across multiple such queries run as a batch. This is because binary operations (such as join) involve the fact table of the $\mathcal{RDW}$ schema. This phenomenon is related to the *problem of multi-query optimization* (MQO). MQO aims to exploit (reuse) results of *common sub-expressions*. This is a major cause of performance problems in database systems [1]. Several research studies were focused on the *modeling* and the *exploitation* of the query interaction. From modeling point of view, several graph structures were proposed: *multiquery graph* [4, 21] and *multiple view processing plan* [22]. It has been exploited by several research studies either in the centralized and the distributed environments by proposing solutions for materializing and caching the intermediate results of the sub-expressions [18, 15, 14, 3].

A. Cuzzocrea and U. Dayal (Eds.): DaWaK 2012, LNCS 7448, pp. 156–167, 2012.

The selection of materialized views is one of the major problems that exploits this phenomenon [22]. When materialized views are selected; their storage may concern two main devices: the *hard disk* and the *main memory*. Materialized intermediate results are usually stored on the disk, especially, when a large number of huge views is selected. Due to the growing size of the main memory, the intermediate results become candidate for bufferisation (consequently stored in the memory). This solution avoids the high cost of reading from or writing to disk. In this case, the MQO impacts directly the buffer management problem (BMP). Another important problem that may be impacted by MQO is the query scheduling (QSP). The QSP consists in finding an ordering of the queries that reduces the overall cost of processing queries. Recently, in [14], the authors propose a method for selecting materialized views incorporating the QSP.

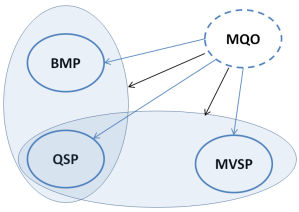

Fig. 1. Interaction between different optimization techniques

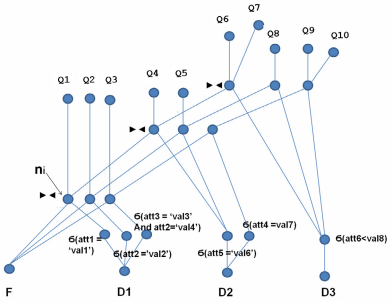

Fig. 2. A MVPP for ten star join queries

By exploring the most important studies related to QSP, we distinguish two main classes: (i) *partial query scheduling*: it consists in finding the best order to evaluate sub-expressions identified by the means of MQO (this problem is known as NP-complete) [17, 18, 20] and (ii) *entire query scheduling*: it considers the entire queries for the scheduling process [20, 14]. Usually, most studies done on the buffer management in traditional databases, in general and in the $\mathcal{RDW}$, in particular, assume that queries are already ordered. As consequence, to execute the first query of a given workload, the DBMS may perform the following tasks: (1) it identifies the relevant disk pages; (2) it loads them in the main memory cache and (3) it executes the query. The next query of the workload may get benefit from the actual content of the buffer if it shares some intermediate results with the first query. If not, DBMS repeats the same process as for the first one, and so on. Based on this scenario, we claim that if the query scheduler has a snapshot of the buffer content, it may reorder the queries to allow them getting benefit from the buffer. This shows the *strong interaction* between the BMP and the QSP [20, 9]. The complexity of the integrated problem including BMP and QSP may be very high [20]. It is proportional to the number of queries of the workload and the number of intermediate sub-expressions. To reduce this

complexity, we propose in this paper to use the *divide and conquer* method. It consists in dividing the queries of the workload into subsets treated separately. This partitioning is performed based on the affinity between queries. To illustrate our proposal, let us consider the following motivating example.

1.1 Motivating Example

We assume a workload with 10 OLAP queries, where each query may be represented by an algebraic tree. Due to the strong interaction between queries, their 10 trees may be merged to generate a graph structure called, *Multiple View Processing Plan* (MVPP) [22] (Figure 3). The leaf nodes of MVPP represent the tables of the $\mathcal{RDW}$. The root nodes represent the final query results and the intermediate nodes represent the common sub-expressions shared by the queries. Among intermediate nodes, we distinguished: (i) *unary nodes* representing the selection (σ) and the projection operations and (ii) *binary nodes* representing join ($\bowtie$), union, intersection, etc. 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 one contains queries having at least one common node. Each cluster is called *hive*. Figure 3 shows the results of clustering of our MVPP, where three hives are obtained. In each hive, we elect a query to be the *Queen-Bee*. Once elected all its common nodes are cached. Thus, queries in the same *hive* will be *ordered* and get benefit from the buffer content. A long this paper, we detail the hints announced in this example: (1) the identification of intermediate nodes, (2) the generation of hives, (3) buffer allocation, (4) query scheduling, and (5) validation of the proposal.

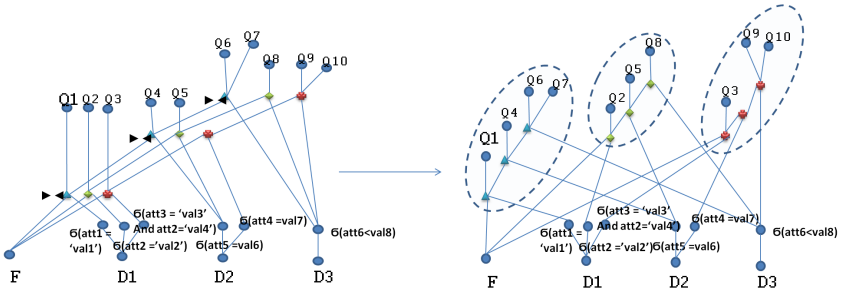

Fig. 3. An example for MVPP with clustered queries (hives)

This paper is structured as follows: Section 2 presents related works. Section 3 formalizes our combined problem and presents all ingredients of our approach. Section 4 details our queen-bee algorithm. Section 5 gives experiments validating our proposal in a simulated environment and on Oracle 11G DBMS. Section 6 concludes our paper by summarizing the main results and presenting some open issues.

2 Related Work

Several studies have shown the importance of the query interaction and its practicability in commercial DBMS [2, 14–16, 21, 22]. This interaction is mainly related to an important problem of the physical design of $\mathcal{RDW}$ known by the MQO. Note that each query can have several alternative evaluation plans, each with a different set of tasks. Therefore, the goal of MQO is to choose the right set of plans for queries which minimizes the total execution time by performing common tasks only once. MQO is known as NP-hard problem [16]. The MQO impacts other hard optimization problems such as materialized view selection problem (MVSP) [22], BMP [7, 6, 8], QSP [5, 12] and the problem combining BMP and QSP [19, 9, 20] as shown in Figure 1.

The BMP got a lot of attention from the database community, where it has studied in different types of databases: (a) traditional databases [7, 6, 8], (b) semantic databases [23], (c) data warehouses [15], (d) flash databases [13]. In the first generation of studies related to BMP, solutions were proposed to allocate pages and to replace them when the buffer is full using policies like *LRU*, *FIFO*, etc. without considering the impact of the MQO. In the second generation, some research efforts were concentrated on incorporating MQO in the buffer allocation [7, 15], where algorithms for selecting common intermediate results to be cached in a limited cache space were proposed.

Similarly, the QSP was studied in isolated way in several environments: *centralized* [20], *distributed* and *parallel* databases/data warehouses [12]. It has been proved as a strongly NP-complete problem [9, 5, 12]. After, it has been mixed with other optimization problems like the MVSP [14]. [19, 9] considered the problem of caching and QSP in the context of $\mathcal{RDW}$ and proposed some heuristics. [20] presented several issues related to the combination of BMP and QPS by considering MQO problem. A complete hardness study was proposed and a tentative of algorithms without a real validation is proposed. By summarizing the few existing studies on combined BMP and QSP (BMQSP) considering MQO, some observations are identified:

- they explore the large search space of the combined problem which may be very large.
- they use simple cost models that ignore parameters related to the cache content, the order of queries, the size of intermediate results of joins, the characteristic of star join queries of $\mathcal{RDW}$, etc.
- they do not validate their proposals using a real DBMS with large memories.

3 Formalization and Cost Models

In this section, we give some key concepts to facilitate the formalization of our problem. We also give a hardness study and describe our cost model. In this work, we consider some assumptions: **(1)** prior knowledge of the workload (offline scheduling), **(2)** a centralized $\mathcal{RDW}$ environment and **(3)** the initial cache content is considered as empty.

3.1 Concepts and Formalization

Given a workload with n queries generated by user applications presented in a queue. Based on their MVPP, some definitions are given (Figure 3).

Definition 1. *The fan-out of a query is defined as the number of overlapping nodes with other queries.*

Definition 2. *The overlapping accumulated cost of a query (noted OAC) is the total execution cost of all its overlapping nodes with other queries. It represents the participation of this query in the overall cost reduction.*

To facilitate the understanding of the formalization of the combined problem BMQSP, we start presenting a separate formalization of both BMP and QSP.

BMP is formalized as follows: **(1)** Inputs : (i) $\mathcal{RDW}$, (ii) a workload with a set of queries $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$ represented by a MVPP, (iii) a set of intermediate nodes of MVPP candidates for caching $\mathcal{N} = \{no_1, no_2, ..., no_l\}$, **(2)** Constraint: a buffer size $\mathcal{B}$ and **(3)** Output : a buffer management strategy $\mathcal{BM}$ that allocates nodes in the buffer to optimize the cost of processing $\mathcal{Q}$.

QSP is formalized as follows: **(1)** Inputs : (i) $\mathcal{RDW}$, (ii) a workload with a set of queries $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$ and (iii) a $\mathcal{B}M$. **(2)** Output : scheduled queries $\mathcal{QS} = \{SQ_1, SQ_2, ..., SQ_n\}$.

BMQSP is described based on the above formalizations as follows:

- Inputs : (i) A $\mathcal{RDW}$ and (ii) a set of queries $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$ represented by a MVPP, (iii) a set of intermediate nodes of MVPP candidates for caching $\mathcal{N} = \{no_1, no_2, ..., no_l\}$;
- Constraint: a buffer size $\mathcal{B}$;
- Output : (i) scheduled queries $\mathcal{SQ} = \{SQ_1, SQ_2, ..., SQ_n\}$ and (ii) a $\mathcal{B}M$, minimizing the overall processing cost of Q.

3.2 Hardness Study

To compute the complexity of our BMQSP, we first relax the buffer management. Exhaustively, a workload with n queries requires $n!$ possible schedules. If we consider the cache, the number of possible subsets of nodes candidates[1] for bufferization is given by: $\sum_{i=1}^{l}(i!)$. Thus, simultaneous resolution of query scheduling and buffer management needs an exploration of a search space of size: $n! \times \sum_{i=1}^{l} i!$, where n and l represent respectively, the number of queries and intermediate results. This is because different solutions may potentially interact with each other. This complexity motivates us to develop efficient and reasonable solutions.

[1] We assume that a node is cached at most once during the workload runtime.

3.3 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 two cases: without caching and with caching.

Without Caching. Most studies using cost models for selecting optimization structures ignore buffer management 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 $\mathcal{D}^{Q_i} = \{D_1^{Q_i}, ..., D_{n_i}^{Q_i}\}$ 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 [10].

We assume that each query Q_i is represented by an ordered set of all the operations of its plan : $op_1^{Q_i}, op_2^{Q_i}, \dots op_{l_i}^{Q_i}/op_k^{Q_i} \in \{FJ, IR, AG\}$ from the first join till the final operation, with l_i the number of operations in Q_i. We define a function $size(op_k^{Q_i})$ estimating the result size of $op_k^{Q_i}$. Recursively, we estimate the cost of a query starting from the final operation $op_{l_i}^{Q_i}$ as follows :

$$Cost(op_k^{Q_i}) = \begin{cases} size(op_k^{Q_i}) & \text{if } k = 1 \\ size(op_k^{Q_i}) + Cost(op_{k-1}^{Q_i}) & \text{if } k \in [2, l_i]\} \end{cases} \quad (1)$$

With Caching. To take into account cache management in our cost model, a buffer content checking is required. The cost of an operation $op_k^{Q_i}$ is ignored if : (i) its result is cached, or (ii) one of its successors ($op_{k+\alpha}^{Q_i}$ with $\alpha > 1$) is cached. For this reason, we define a function $b(op_k^{Q_i})$ to check if the result of $op_k^{Q_i}$ is present in the buffer (=1) or not (=0). Therefore, the execution cost will be :

$$Cost(op_k^{Q_i}) = \begin{cases} \left[size(op_k^{Q_i})\right] \times \prod_{j=k}^{l_i} b(op_j^{Q_i}) & \text{if } k = 1 \\ \left[size(op_k^{Q_i}) + Cost(op_{k-1}^{Q_i})\right] \times \prod_{j=k}^{l_i} b(op_j^{Q_i}) & \text{if } k \in [2, l_i] \end{cases} \quad (2)$$

Total Cost. The total execution cost of the workload is estimated as the sum of all queries cost: $Total\_cost = \sum_{i=1}^{n} Cost(op_{l_i}^{Q_i})$.

4 Queen-Bee Algorithm

To reduce the complexity of the combined problem, we propose an approach inspired from the natural life of bees. It is illustrated in Figure 4. 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. We associate the dynamic buffer management strategy DBM[11] 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[2] 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 the buffer content. To do so, three modules define our queen-bee algorithm: (1) generating of a query graph with connected components (QGCC), (2) sorting the components (α-sort) which is optional depending on whether queries have priority or not and (3) sorting queries inside each component (β-sort). Contrary to the scheduling strategy proposed in [11] (called dynamic query scheduler) that takes into account only the cache content, our scheduler considers other parameters regarding the *query execution cost*, the *query fan-out* and the overlapping accumulated cost of a query (OAC) (see Definitions 1 and 2). These factors are used to sort nodes of each *hive*.

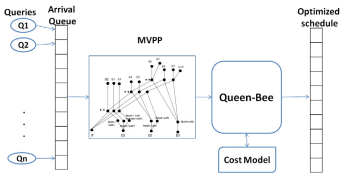

Fig. 4. Our methodology

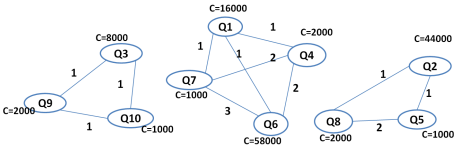

Fig. 5. An example of QGCC

Generating the Query Graphs with Connected Components. Starting from MVPP, a QGCC representing *hives* is obtained. 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.

α-Sort. In our study, all queries have the same priority. Therefore, the required data for two different hives are disjoint ({(Q1,Q7,Q4,Q6);(Q3,Q9,Q10);(Q2,Q8,Q5)} and {(Q3,Q9,Q10);(Q1,Q7,Q4,Q6);(Q2,Q8,Q5)} have the same cost).

β-Sort. The queries inside each component need to be scheduled. For this reason, β-sort takes each component and schedules its queries by performing two steps: (1) identification of the *Queen-Bee* based on the chosen criterion (cost, fan-out, *OAC*)[3] (2) exploration of the rest of nodes (using the same criterion used

[2] We define the rank value of a node no_i as a counter representing the number of queries accessing no_i.

[3] The choice of criterion is done by the database administrator.

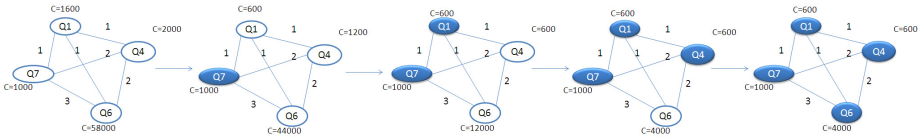

Fig. 6. Traversing the component by the minimal cost

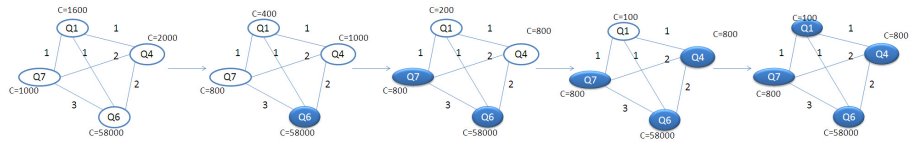

Fig. 7. Traversing the component by the maximal fan-out

for selecting the queen-bee), and (3) once the connected component is entirely traversed, return the obtained sub-schedule. Figure 6 gives an example of β-sort 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.

The fan-out criterion may also be interesting for choosing the Queen-Bee. The idea is to start by the query giving more relevant data to others. The figure 7 gives an example of β-sort of fan-out in descending order. We can easily observe that theoretically, sorting by minimal cost gives different performance than sorting by maximal fan-out [4]. Sorting criteria are studied in the experimental studies.

5 Experiments

To validate our proposal, we conduct intensive experimental studies in both *theoretical* and *effective* ways. The theoretical experiments are based on our mathematical cost model and our simulation tool. The results by our simulations are then implemented on Oracle11G.

5.1 Dataset and Workload

Our experiments are done on the Star Schema Benchmark (SSB of 100GB) having a fact table *Lineorder* (6,000,000 $\times SF$ tuples) and 4 dimension tables

[4] In Figure 6, total cost=1000+600+600+4000=6200 I/O, and in Figure 7, total cost=58000+800+800+100=59700 I/O.

with a scale factor $SF = 100$. A server with 32 GB of RAM is used. We consider thirty queries, detailed in [10], covering different types of OLAP queries.

5.2 Simulation Tool

To make easier our testing, we developed a *simulation tool* using a Java development environment. It contains three main modules : **(1)** $\mathcal{RDW}$ connectivity and meta-data extraction needed for the cost model, **(2)** Setting $\mathcal{RDW}$ parameters and handle the workload and **(3)** 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 optimization module gives them the possibility to tune some parameters. Otherwise, administrators deploy them on Oracle11G using appropriate scripts.

5.3 Obtained Results

We start by studying the impact of criteria used to select the queen-bees of all components: *cost*, *fan-out* and OAC. We run experiments by considering ascendant sorting (minimal value) and descend sorting (maximal value) for each criterion. Figure 8 summarizes the obtained results. The three factors give better performance when they are used in an ascendant sorting. Note that a query with a minimal *fan-out* or OAC is usually a query with few nodes (less operations). This explains the fact that these two factors give similar results.

In [11], we proposed some algorithms to get a near optimal query scheduling and its buffer management using respectively dynamic query scheduler (DQS) and dynamic buffer management (DBM). In Figure 10, 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: (1) DBM is more adapted than LRU in our context; (2) DQS evolves the efficiency of the buffer management policy; and (3) Queen-Bee gives the same performance as the heuristics that use 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 and Queen-Bee at a buffer space of $32GB$.

One of the motivations of our proposal is to prune the search space. In Figure 9, we can see that the queen-bee algorithm is much faster than the DBM-DQS even though they give the same query performance.

5.4 Validation on Oracle11g

For validation, we deploy our simulation results on an Oracle 11g 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

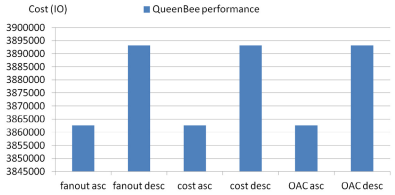

Fig. 8. Different factors used in β-sort

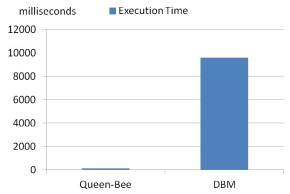

Fig. 9. Comparing Run time

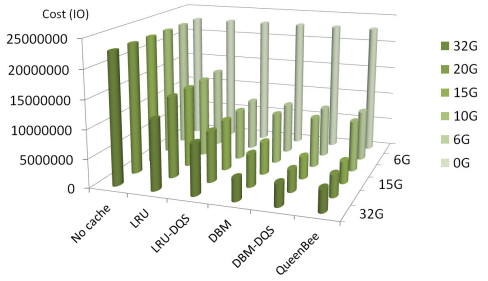

Fig. 10. Comparing different algorithm's performance

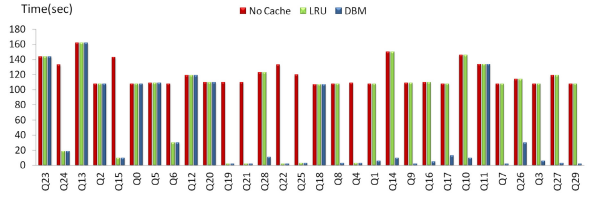

Fig. 11. LRU vs DBM : experiments with static schedule

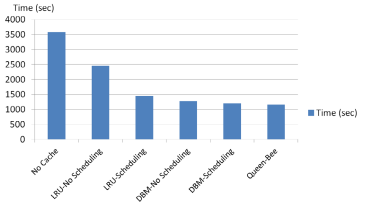

Fig. 12. Deploying algorithms' results

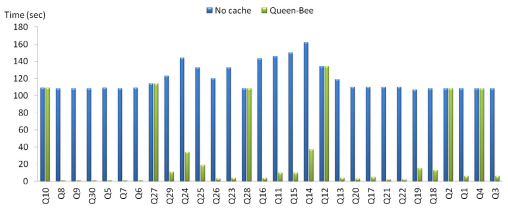

Fig. 13. Deploying Queen-Bee results

algorithms. The DBMS parameters are tuned to prepare buffer pool[11]. To compare LRU with the DBM, we take the same workload in a static order on Oracle 11g with 6GB of buffer pool. The queries are executed using: (1) no buffer management, (2) LRU, and (3) our DBM. In Figure 11, 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.

The validation of the simulation experience in Figure 10 is done on our DBMS. Real performance is given in Figure 12 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 can see 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.

6 Conclusion

The query interaction is one of the most important characteristics of data warehouse applications. It has been exploited to solve several complex optimization problems during the physical design phase of a data warehouse. Recently, it has been exploited to solve problems combining other sub-problems such as buffer management and query scheduling. This integrated problem is studied in this paper. To tackle this problem, a methodology is given. It starts from a set of queries obtained from user applications; and placed in a queue. Then, algebraic trees of these queries are merged to form a graph, called multiple views processing plan. The intermediate nodes of this plan are candidates for bufferization. To avoid the explosion of evaluating each intermediate node and study its effect on query scheduling, we propose an approach inspired from the natural life of bees. It partitions queries into clusters (hives), and for each hive, a query is elected (called queen bee). Once it is elected, all its nodes are potentially cached in the buffer. Based on sorting mechanism, each hive is scheduled. This sorting is based on three main criteria: the query cost, query fan-out and the overlapping accumulated cost. Experimental studies were conducted by the use of a simulator and its results are validated on Oracle11G. The obtained results are encouraging and show the effectiveness of our queen-bee approach.

Actually, we are extending this work to handle queries defined on semantic data warehouses. An interesting issue that should be considered concerns the impact of the buffer management and the query scheduling on the horizontal partitioning problem.

References

1. Ahmad, M., Aboulnaga, A., Babu, S., Munagala, K.: Modeling and exploiting query interactions in database systems. In: Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM), pp. 183–192 (2008)

2. Ahmad, M., Aboulnaga, A., Babu, S., Munagala, K.: Interaction-aware scheduling of report-generation workloads. VLDB Journal, 589–615 (2011)
3. Arion, A., Benzaken, V., Manolescu, I., Papakonstantinou, Y.: Structured materialized views for xml queries. In: VLDB, pp. 87–98 (2007)
4. Chakravarthy, U.S., Minker, J.: Multiple query processing in deductive databases using query graphs. In: VLDB, pp. 384–391 (1986)
5. Chipara, O., Lu, C., Roman, G.-C.: Real-time query scheduling for wireless sensor networks. In: RTSS, pp. 389–399 (2007)
6. Chou, H.-T., DeWitt, D.J.: An evaluation of buffer management strategies for relational database systems. In: VLDB, pp. 127–141 (1985)
7. Cornell, D.W., Yu, P.S.: Integration of buffer management and query optimization in relational database environment. In: VLDB, pp. 247–255 (1989)
8. Effelsberg, W., Härder, T.: Principles of database buffer management. ACM Trans. Database Syst. 9(4), 560–595 (1984)
9. Gupta, A., Sudarshan, S., Viswanathan, S.: Query scheduling in multi query optimization. In: IDEAS, pp. 11–19 (2001)
10. Kerkad, A., Bellatreche, L., Geniet, D.: Heuristics for solving the integrated buffer management and query scheduling problem. Technical report, LIAS-ENSMA (2011)
11. Kerkad, A., Bellatreche, L., Geniet, D.: Simultaneous resolution of buffer allocation and query scheduling problems. In: Proceedings of the 6th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS) (2012)
12. Märtens, H., Rahm, E., Stöhr, T.: Dynamic Query Scheduling in Parallel Data Warehouses. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 321–331. Springer, Heidelberg (2002)
13. Ou, Y., Härder, T., Jin, P.: CFDC: A Flash-Aware Buffer Management Algorithm for Database Systems. In: Catania, B., Ivanović, M., Thalheim, B. (eds.) ADBIS 2010. LNCS, vol. 6295, pp. 435–449. Springer, Heidelberg (2010)
14. Phan, T., Li, W.-S.: Dynamic materialization of query views for data warehouse workloads. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 436–445 (2008)
15. Roy, P., Ramamritham, K., Seshadri, S., Shenoy, P., Sudarshan, S.: Don't trash your intermediate results, cache 'em. CoRR
16. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. 13(1), 23–52 (1988)
17. Sethi, R.: Complete register allocation. In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, pp. 182–195 (1973)
18. Swami, A.N., Gupta, A.: Optimization of large join queries. In: SIGMOD Conference, pp. 8–17 (1988)
19. Tan, K.-L., Lu, H.: Workload scheduling for multiple query processing. Information Processing Letters 55(5), 251–257 (1995)
20. Thomas, D., Diwan, A.A., Sudarshan, S.: Scheduling and caching in multiquery optimization. In: COMAD, pp. 150–153 (2006)
21. Le, W., Kementsietsidis, A., Duan, S., Li, F.: Scalable multi-query optimization for sparql. In: Proceedings of the International Conference on Data Engineering, ICDE (2012)
22. Yang, J., Karlapalem, K., Li, Q.: Algorithms for materialized view design in data warehousing environment. In: VLDB, pp. 136–145 (1997)
23. Yang, M., Wu, G.: Caching intermediate result of sparql queries. In: WWW (Companion Volume), pp. 159–160 (2011)