Approaches to Model Query Interactions

Manoel Siqueira[1], José Maria Monteiro[1], Angelo Brayner[2],
José Macedo[1], Javam Machado[1]

[1] Universidade Federal do Ceará - Brasil
{manoeljr, monteiro, jose.macedo, javam}@lia.ufc.br
[2] Universidade de Fortaleza - Brasil
brayner@unifor.br

Abstract. A typical database workload consists of several query instances of different query types running concurrently. The execution of each query may interact with the execution of the other queries. It is well known that such interactions can have a significant impact on database system performance. In this article we propose three new approaches to model and measure query instance and query type interactions. Our approaches require no prior assumptions about the internal aspects of the database system, making it non intrusive, namely, portable across systems. Furthermore, to demonstrate the profit of exploiting query interactions, we have developed a novel interaction-aware query scheduler for online workloads, called Intelligent Scheduler for Multiple-query Execution Ordering (ISO, for short). In order to verify the efficiency of the proposed approaches for measuring query interaction and of ISO, an experimental evaluation using TPC-H workloads running on PostgreSQL has been done. The results show that the proposed approach has potential to improve the efficiency of database tuning tools.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous

Keywords: query interactions, query scheduling, interaction factor

1. INTRODUCTION

A typical database workload consists of a mix of multiple query instances of different query types that run concurrently and interact with each other. A query type can be defined as a template for SQL queries, which consists of a SQL expression with parameter markers. Whenever a template is instantiated with a set of parameter values, one has a query instance. For example, the popular TPC-H decision support benchmark [TPC 2013] defines 22 query templates. From each TPC-H query template, several query instances can be generated. From now on, we use the term query type to represent a query template. Thus, there are 22 query types in TPC-H benchmark.

Figures 1a and 1b illustrates the notions of query type and query instance. Figure 1a shows a TPC-H query type Q_j with one parameter marker, represented by the symbol "?". Different value settings of the parameter marker yield different instances of the query type. In turn, Figure 1b illustrates a query instance q_{j_1} of the query type Q_j.

Query instances in a workload can have interactions with a significant impact on database performance, which can be positive or negative [Ahmad et al. 2009]. Ahmad et al. [2009] and Ahmad et al. [2011] show many interesting examples of the effect of query interactions on database performance. However, very little work in the database literature deals with the problems of modeling and measuring query interactions [Ahmad et al. 2009].

In this work, we propose three new approaches to model and measure query instance and query type interactions. The proposed approaches, denoted intercalation strategy (IS), data retrieving rate (DRR) and greedy two-dimensional array (GBA), do not require any prior assumptions on internal

This work was partially funded by FUNCAP and CNPq.

```
1 SELECT *
2 FROM lineitem AS l, orders AS o,
3   supplier AS s, nation AS n
4 WHERE l.l_orderkey = o.o_orderkey AND
5   l.l_suppkey = s.s_suppkey AND
6   s.s_nationkey = n.n_nationkey AND
7   n.n_name = ?;
```

(a) Query type (template)

```
1 SELECT *
2 FROM lineitem AS l, orders AS o,
3   supplier AS s, nation AS n
4 WHERE l.l_orderkey = o.o_orderkey AND
5   l.l_suppkey = s.s_suppkey AND
6   s.s_nationkey = n.n_nationkey AND
7   n.n_name = 'USA';
```

(b) Query instance

Fig. 1: Query examples

aspects of the database system, which makes them non intrusive and consequently portable to existing database systems. The proposed approaches have different inputs and preprocessing overhead. The IS approach is based only on the SQL statements and query execution plans. No preprocessing is needed. The DRR approach is based on the data retrieving rate from hard disk and requires a low level of preprocessing. GBA approach uses a preprocessed two-dimensional array with estimations about the gain reached by the execution of each pair of query instances (or query types).

Based on the approaches to measure query interactions, an intelligent scheduler for multiple-query execution ordering, denoted ISO, has been implemented. ISO is interaction-aware query scheduler for online workloads. Thus, given a set $Q = \{q_1, q_2, ..., q_n\}$ of queries, ISO is able to dynamically define the most efficient execution order for the queries belonging to Q.

The remainder of this article is organized as follows. Section 2 discusses related work. Section 3 introduces the proposed approaches to model and measure query interactions. Section 4 presents the proposed query scheduler. Thereafter, Section 5 presents and discusses the results of the executed experiments. Finally, Section 6 concludes the article.

2. RELATED WORK

There is little work discussing query interaction and efficient query execution order. In this sense, Ahmad et al. [2009] proved that query interactions have a significant impact on database performance. However, they did not discuss how to model and measure query interactions.

Ahmad et al. [2008] and Ahmad et al. [2011] describe an experiment-driven modeling for batch scheduling. Also, they propose an online scheduling algorithm based on a metric called NRO (Normalized Runtime Overhead). To calculate NRO, a time consuming preprocessing phase is used. In this step, i query sets, denoted mixes, are constructed and run. Each query mix m_i has $N_{i,j}$ instances of each query type j. O'Gorman et al. [2005] describe a scheduling algorithm based on a two-dimensional array which stores in each cell $c_{i,j}$ a rate for executing query instance q_i before q_j. All these approaches need a heavy preprocessing to be used and can be inappropriate when each query has a deadline.

Some work are intrusive and use the concurrent query execution scenario to get performance improvements [Roy et al. 2000; Tan and Lu 1995]. These approaches make changes in the DBMS query optimizer to explore some properties and reuse common data among queries that are stored in memory. Other work are based on query optimizer estimations to support workload management decisions, including scheduling ones [Niu et al. 2007; Niu et al. 2009]. However, this cost does not help so much when queries have a deadline to fulfill.

This article extends previous work in several aspects: *(i)* it provides three new approaches to model and measure query interactions; *(ii)* the proposed approaches present different preprocessing overhead levels; *(iii)* no changes in the DBMS query optimizer are needed and *(iv)* optimizer estimations for query execution cost are not used. In fact, our solution is based on query execution plans, but it does not care about their estimations because this information is not good enough to indicate if a query can execute in an acceptable time.

3. MEASURING AND MODELING QUERY INTERACTIONS

Nowadays, most databases are stored in hard disks. Data access rates in hard disks are several magnitude orders lower than in main memory, specially w.r.t. random accesses.

To execute a query q_i, the buffer manager may load data pages into the buffer pool, which are used by another concurrently running query q_k. Such a scenario characterizes a query interaction between q_i and q_k. Of course, executing q_i before q_k (or vice-versa), q_k can profit from the fact that pages, necessary to process it, are already in the buffer pool. Based on this observation, this article presents three approaches to model and measure query instance and query type interactions. In this section we will discuss these approaches in detail.

The proposed approaches uses the concept of *interaction factor*, which is a number between 0 and 1. The interaction factor quantifies the interaction between two query instances or between two query types. Values close to 1 indicate strong interaction while values close to 0 indicate weak interaction.

3.1 Table Relationships

The IS (Subsection 3.2) and DRR (Subsection 3.3) approaches computes the interaction factor between two queries by means of the notion of table relationship. There are two types of table relationships: *possible intersection* and *disjunction*.

Let q_i and q_j be two query instances, P_{q_i} and P_{q_j} the query execution plans for q_i and q_j (generated by the DBMS's query processor). Consider that T_i is a set with all tables $t_i \in P_{q_i}$. Given a table t_i, such that $t_i \in P_{q_i}$, O_{t_i} is the set of table operations on t_i in P_{q_i} (for example, *table scan*, *index scan* or *index seek*). Thus, o_{t_i}, where $o_{t_i} \in O_{t_i}$ is a table operation on t_i in P_{q_i}.

So, given two tables t_i and t_j, the relationship between t_i and t_j is possible interaction if and only if: (i) $t_i = t_j$ and (ii) $\exists o_{t_i} \in O_{t_i}$ and $\exists o_{t_j} \in O_{t_j}$ such that one of the following conditions occur: (a) o_{t_i} or o_{t_j} is a *table scan* operation; (b) o_{t_i} and o_{t_j} are *index scan* using distinct columns, or; (c) o_{t_i} and o_{t_j} are *index scan* using the same columns and there are overlaps on index search keys. If one of these conditions occurs, o_{t_i} and o_{t_j} may access common data.

On the other hand, the table relationship between t_i e t_j is *disjunctive*, if: *(i)* $t_i \neq t_j$, or; *(ii)* $t_i = t_j$ and none of the following conditions occurs: (a) o_{t_i} or o_{t_j} is a *table scan* operation; (b) o_{t_i} and o_{t_j} are *index scan* using distinct columns, and; (c) o_{t_i} and o_{t_j} are *index scan* using the same columns and there are overlaps on index search keys. If neither of these conditions occur, then o_{t_i} and o_{t_j} do not access common data.

3.2 Intercalation Strategy (IS)

The Intercalation Strategy is based only on the SQL statements and query execution plans. No preprocessing is needed. Algorithm 1 describes the steps of this approach to compute interaction factor.

Algorithm 1 has as input the following parameters: two query instances q_{j_1} and q_{l_2}, where we assume that q_{j_1} starts its execution before q_{l_2}, and; a DBMS driver (*dbms*), which is responsible for accessing database catalog (metabase). The algorithm output is the interaction factor between q_{j_1} and q_{l_2}, denoted by $f_{q_{j_1},q_{l_2}}$. It's important to note that $f_{q_{j_1},q_{l_2}}$ can be different from $f_{q_{l_2},q_{j_1}}$. According to the notation used in Subsection 3.1, $T_{q_{j_1}}$ represents the set of tables belonging to the query execution plan $P_{q_{j_1}}$ and $T_{q_{l_2}}$ is the set of tables belonging to $P_{q_{l_2}}$. The variable $size(t_k)$ represents the size of the table t_k (in Kilobytes), $totalSize$ stores the sum of $size(t_k)$ for each $t_1 \in T_{q_{j_1}}$ and $t_2 \in T_{q_{l_2}}$.

The *tableRelationship* variable represents table relationship (possible interaction or disjunction) between two tables. From line 8 to 25 the α variable is calculated. Finally, in line 26, the interaction factor (α) is returned. During the tests W_1 and W_4 were set to 0.5, W_2 to 1 and W_3 to 0.25. These values were chosen to benefit occurrences of table scan in t_1 and index seek in t_2.

Algorithm 1: Intercalation strategy

```
   input  : q_{j1}, q_{l2}, dbms
   output : interaction factor
1  begin
2      if isNull(q_{j1}) or isNull(q_{l2}) then
3          return 0;
4      end if
5      T_{q_{j1}} ← dbms.parseQuery(q_{j1});
6      T_{q_{l2}} ← dbms.parseQuery(q_{l2});
7      totalSize ← getTotalSize(T_{q_{j1}}, T_{q_{l2}});
8      α ← 0;
9      foreach t1 in T_{q_{j1}} do
10         foreach t2 in T_{q_{l2}} do
11             tableRelationship ← t1.getRelationship(t2);
12             if isPossibleIntersection(tableRelationship) then
13                 if t1.hasFullScan() and t2.hasFullScan() then
14                     auxWeight ← W1;
15                 else if t1.hasFullScan() and not t2.hasFullScan() then
16                     auxWeight ← W2;
17                 else if not t1.hasFullScan() and t2.hasFullScan() then
18                     auxWeight ← W3;
19                 else
20                     auxWeight ← W4;
21                 end if
22                 α ← α + auxWeight × size(t1) / totalSize;
23             end if
24         end foreach
25     end foreach
26     return α;
27 end
```

The key idea of the IS approach is to put closer queries with tables in common in the query execution schedule. The access method is important because the chance of improving the performance by executing a query q with an index seek operation on a table t after another query q' which accesses the same table t by means of a table scan operation is higher than executing q' before q.

3.3 Data Retrieving Rate (DRR)

DRR approach relies on the following parameters: SQL statements, query plans and data retrieving (access) rate for each query type. For that reason, a preprocessing step is required. This preprocessing consists of running, for each query type Q_j, a set S_j of n query instances m times. The average of the data retrieving rate for the instances in S_j with m running iterations is the DRR for the query type Q_j. In the experiments, we have fixed $m = 5$ and evaluated two values for n: $n = 1$ and $n = 5$.

Actually, DRR represents the amount of data transferred from the disk to the database buffer pool (main memory) per time unit (KB/s in our tests). Thus, it is possible to infer that if a query type has a low DRR its query instances will access data in a low rate as well. By doing this, the following heuristic is applied: query instances with greater DRRs should be executed before query instances with low DRR.

Algorithm 2 describes the steps implemented by the DRR approach. Algorithm 2 has as input parameters: two query instances q_{j_1} and q_{l_2} (q_{j_1} starts its execution before q_{l_2}); the query types Q_j and Q_l of q_{j_1} and q_{l_2}, respectively; an array R with the computed DRR for each query type; and a DBMS driver (*dbms*) responsible for accessing database catalog (metabase). The algorithm output is the interaction factor between q_{j_1} and q_{l_2}, denoted by $f_{q_{j_1},q_{l_2}}$. In Algorithm 2, $max(R)$ represents highest value in the array R and R_{Q_k} is the DRR for Q_k. Lines 2 to 7 do the same as lines 2 to 7 from Algorithm 1. From line 8 to 22, the value of the interaction factor α is calculated. This process starts with the initialization of this variable in line 8. Next, it is verified the relationship between each pair of tables computed in the nested loops, which is depicted from line 9 to 22. Whenever possible intersection table relationships are found (line 12), the value of α is modified (line 19) based

Algorithm 2: DRR approach

```
   input  : q_{j1}, q_{l2}, R, Q_j, Q_l, dbms
   output : interaction factor
1  begin
2      if isNull(q_{j1}) or isNull(q_{l2}) then
3          return 0;
4      end if
5      T_{q_{j1}} ← dbms.parseQuery(q_{j1});
6      T_{q_{l2}} ← dbms.parseQuery(q_{l2});
7      totalSize ← getTotalSize(T_{q_{j1}}, T_{q_{l2}});
8      α ← 0;
9      foreach t_1 in T_{q_{j1}} do
10         foreach t_2 in T_{q_{l2}} do
11             tableRelationship ← t_1.getRelationship(t_2);
12             if isPossibleIntersection(tableRelationship) then
13                 if t_1.hasFullScan() then
14                     auxWeight ← W_1 × getNormalizedDRR(Q_l);
15                 else
16                     auxWeight ← W_2 × getNormalizedDRR(Q_l);
17                 end if
18                 size(t_1) ← dbms.size(t_1.getTableName());
19                 α ← α + auxWeight × size(t_1) / totalSize;
20             end if
21         end foreach
22     end foreach
23     return α;
24 end
```

on the set O_{t_1} (line 13 – for more details, see Subsection 3.1), the constants W_1 or W_2 (depending on set O_{t_1}, as showed between lines 13 to 17), DRR of Q_l (lines 14 and 16) and the analyzed table weight (represented by $\frac{size(t_1)}{\text{totalSize}}$, in line 19). From line 19, α variable is modified. From this line, the *auxWeight* variable refers to interaction caused by table operations of t_1 (when $t_1 = t_2$) in q_{j_1} and q_{l_2}, whereas $\frac{size(t_1)}{\text{totalSize}}$ represents its impact on interaction between these two queries.

It is important to observe that the DDR approach has the same table relationship concept as IS. Moreover, the DDR and IS approaches are based on the heuristic in which queries accessing tables by table scans load more data to the buffer pool, allowing thus more data page reuse. For that reason, during the experiments, W_1 was set to 1 and W_2 to 0.5. Nevertheless, in the DDR approach, the value of the interaction factor α should incorporate the data retrieving rate. This is because DDR approach implements the following heuristic: query instances with greater DRRs should be executed before query instances with low DDRs. The DRR parameter is used in its normalized form, computed by $getNormalizedDRR(Q_l)$, which basically consists of $\frac{R_{Q_l}}{max(R)}$.

3.4 Greedy with Two-Dimensional Array (GBA)

The GBA approach is based on a two-dimensional array obtained by a preprocessing step, which is described in Algorithm 3. Each line j of the two-dimensional array built by GBA represents a query type Q_j and each column l represents a query type Q_l. Hence, a cell $c_{j,l}$ stores the gain of running a query instance of query type Q_l after an instance of query type Q_j. This gain, denoted by $gain(Q_j, Q_l)$, is calculated from the following formula: $rt(Q_j, Q_l)$ - $rt(Q_l)$, where $rt(Q_j, Q_l)$ is the response time for an instance of Q_l that starts its execution immediately after finishing an instance of Q_j and $rt(Q_l)$ is the response time for an instance of Q_l running with free memory.

The input parameter for Algorithm 3 are: a set D of query instances used to calculate the $gain(Q_j, Q_l)$; the number of query types (parameter *typeQty*); the number n of query instances for each query type; the total number m of iterations for each pair of query instances to be run; and a DBMS driver (*dbms*) responsible for accessing database catalog (metabase). In the experiments, we have fixed $m = 5$ and evaluated two values for n: $n = 1$ and $n = 5$.

Algorithm 3: GBA preprocessing

```
   input  : D, typeQty, n, m, dbms
   output : G
1  begin
2      T ← createArray(typeQty);
3      G ← createArray(typeQty, n);
4      for Q_j ← 1 to typeQty do
5          time_{j_1} ← 0;
6          for Q_l ← 1 to typeQty do
7              time_{l_2} ← 0;
8              for i_{j_1} ← 1 to n do
9                  q_j ← D_{Q_j,i_{j_1}};
10                 for i_l ← 1 to n do
11                     q_l ← D_{Q_l,i_l};
12                     for k ← 1 to m do
13                         freeMemory();
14                         time_{j_1} ← time_{j_1} + dbms.runQuery(q_j);
15                         time_{l_2} ← time_{l_2} + dbms.runQuery(q_l);
16                     end for
17                 end for
18             end for
19             G_{Q_j,Q_l} ← time_{l_2} / (n^2 × m);
20         end for
21         T_{Q_j} ← time_{j_1} / (typeQty × n^2 × m);
22     end for
23     for Q_j ← 1 to typeQty do
24         for Q_l ← 1 to typeQty do
25             G_{Q_j,Q_l} ← T_{Q_j} − G_{Q_j,Q_l};
26         end for
27     end for
28     return G;
29 end
```

The first steps of Algorithm 3 are responsible for creating and initializing the arrays T and G. In order to illustrate how Algorithm 3 works, consider that Q_j and Q_l are query types currently being analyzed and that query instances from Q_j are executed before query instances of Q_l. Between lines 5 and 21, the execution time for Q_j is calculated. This is obtained by computing the average of response time of all Q_j query instance execution. From line 7 to line 19 it is computed the execution time for Q_l when running after query instances of Q_j. Thereafter, the gain matrix is built (from lines 23 to 27). In line 28, the gain matrix G is returned.

Algorithm 4: GBA approach

```
   input  : Q_j, Q_l, G
   output : interaction factor
1  begin
2      α ← 0;
3      if not isNull(Q_j) and not isNull(Q_l) then
4          α ← G_{Q_j,Q_l} / max(G);
5      end if
6      return α;
7  end
```

After the two-dimensional array GBA is built, Algorithm 4 is triggered. According to the steps of Algorithm 4, in the same way as previous approaches, α is initialized with value 0, as it is showed in line 2. Next, in line 3, it is verified if Q_j and Q_l are defined types (not null). In this case, in line 4, the performance gain is read in the cell G_{Q_j,Q_l} and α (the interaction factor) receives $\frac{G_{Q_j,Q_l}}{max(G)}$, where $max(G)$ is the maximum value stored in G, Q_j is a line of G and Q_l is a column of G. So, the value of the interaction factor is normalized ($[0, 1]$). Finally, the interaction factor is returned in line 6.

4. ISO: AN INTERACTION-AWARE QUERY SCHEDULER

Algorithm 5: Interaction-aware query scheduling

```
   input    : S, q_new, q_last, dbms
1  begin
2      bestPosition ← 1;
3      Δ_factor_max ← -1;
4      previousQuery ← q_last;
5      for i ← 1 to S.getLength() do
6          nextQuery ← S_i;
7          Δ_factor ← getFactor(previousQuery, q_new, dbms)
8                     + getFactor(q_new, nextQuery, dbms)
9                     - getFactor(previousQuery, nextQuery, dbms);
10         if Δ_factor ≥ Δ_factor_max then
11             bestPosition ← i;
12             Δ_factor_max ← Δ_factor;
13         end if
14         previousQuery ← nextQuery;
15     end for
16     Δ_factor ← getFactor(previousQuery, q_new, dbms);
17     if Δ_factor ≥ Δ_factor_max then
18         bestPosition ← S.getLength() + 1;
19     end if
20     S.add(bestPosition, q_new);
21 end
```

Based on the proposed approaches to model and measure query interactions (IS, DRR and GBA), we have developed a interaction-aware query scheduler for online (and batch) workloads, denoted ISO. ISO uses the interaction factor to define the query execution order. The scheduler goal is to optimize workload query response time by dynamically choosing, for each query instance, the position in the scheduling queue which provides the highest interaction factor gain.

The steps implemented by ISO are showed in Algorithm 5. The Algorithm 5 requires four input parameters: the scheduling queue S; the query instance to be scheduled q_{new}; the last query instance q_{last}, which left the queue S, and; the database driver *dbms*. The first steps executed by Algorithm 5 consist of the initialization of the following variables (lines from 2 to 4): *bestPosition*, which represents the position for q_{new} in S; $\Delta_{factor_{max}}$, containing the greatest interaction gain, and; *previousQuery*, which stores the position in S of the previous query if q_{new} has to be added to current position in S. In line 6, variable *nextQuery* is set to *previousQuery*. Between lines 5 and 19, q_{new} is tested in each position of S. Let i be a position of S and S_i a query added in the position i of S. From line 9, it is calculated the interaction factor gain Δ_{factor} of adding q_{new} in the position i of S, namely, between queries represented by variables *previousQuery* and *nextQuery*. In case of Δ_{factor} being the greatest interaction factor gain for while, the *bestPosition* variable and the best interaction factor gain $\Delta_{factor_{max}}$ are updated (lines 11 and 12, respectively). Lines from 16 to 19 are similar to lines from 9 to 13. The main difference is the current position tested. In the first case, it is a position i of S where there is a query added, whereas in the second case, the query q_{new} is tested in the first free position of S. The last step of Algorithm 5 is add the query instance q_{new} in the position of S which provides the greatest estimated global gain obtained by the formula depicted in line 10.

5. EVALUATION

In order to evaluate the proposed three approaches to model and measure query interactions and ISO scheduler, we have chosen a scenario in which queries are executed in a sequential way. The choice for such a scenario was motivated by the discussion about the need of predictability in the current paradigm of optimization problem [Florescu and Kossmann 2009]. The idea is to ensure that the queries are executed in an acceptable time, but not as quick as possible. According to this approach, optimizing for the 99 percentile is more important than for the average. Thus, for instance, in the case that each query has its deadline, a solution that is optimized by 99 percentile is more appropriate.

Chi et al. [2011] show one example of this, that describes a solution based on agreements called SLAs (Service Level Agreements) with a specific metric: query response time. In this case, each query has to be run in a defined period of time to generate profit to the service provider, otherwise penalties can be applied.

When each query has a maximum time to be run, the use of a concurrent solution can be an inefficient way to schedule queries because of the complexity in predicting their response times if compared with sequential approaches. Many factors such as operating system scheduling algorithm and resource utilization impact of each query during the execution interfere in concurrent solution effectiveness. In fact, they can decrease the workload response time, but it is harder than in sequential solutions to control the query deadline fulfillment or to guarantee the profit in an SLA environment that uses query response time as metric, for example. Therefore, a sequential solution shows to be more feasible in some situations than a concurrent one.

5.1 Experimental Setup

A 2 GHz Intel Core 2 Duo machine with 3 GB of RAM and 500 GB of HD, using Ubuntu 11.10 of 64 bits as operating system, has been used to execute the experiments. The tests were run implementing the TPC-H benchmark with scale factor of 2 GB, using DBGEN/DBT-3 project, in PostgreSQL 9.1.3 database system.

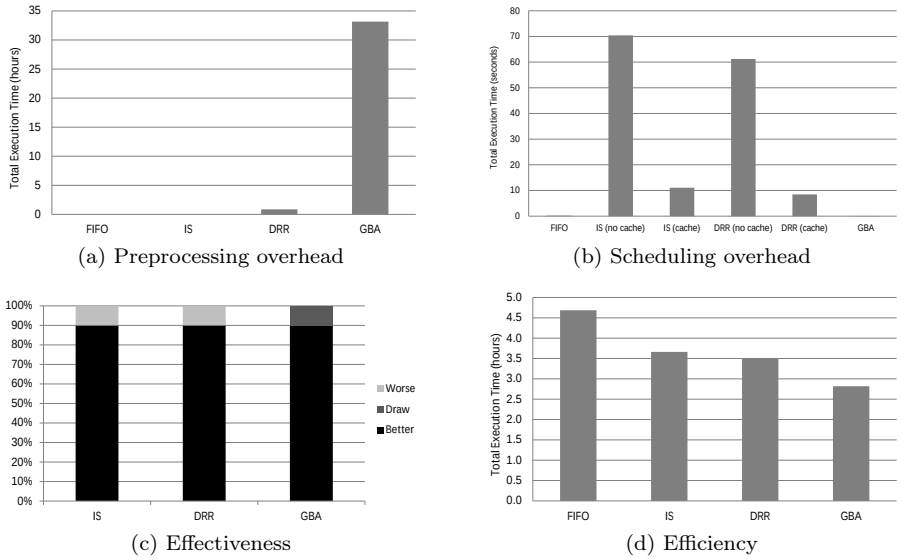
The experiments have been conducted by using two different workloads, based on the number of queries to be scheduled. In the first workload, one query instance has been created for each TPC-H query type, totaling 22 queries. These instances were generated setting different values for the template parameters, using QGEN/DBT-3. Thereafter, ten different orderings have been randomly yielded, each of which with 22 query instances. For this workload, the DRR vector, used by DRR approach, and gain matrix G, used by GBA approach, were created using $n = 1$. In the second workload, five query instances have been created for each TPC-H query type, totaling 110 queries. For this second workload, the DRR vector, used by DRR approach, and gain matrix G, used by GBA approach, were created using $n = 5$. Furthermore, twenty different ordering have been randomly produced. It is important to note that the different orderings simulate the orders in which the queries are arriving to be executed. For each defined ordering, a FIFO strategy is used to send queries to ISO.

5.2 Test Results

In this section, we present the results of the experiments we carried out. For that, we used four metrics: preprocessing-step execution time; execution time for computing the query-execution schedule; effectiveness and efficiency. Since the DRR and GBA approaches require a preprocessing step, we have used preprocessing-step execution time metric. Figure 2a depicts the results regarding that metric for the first workload (22 query instances and 10 random orderings). In turn, Figure 3a brings the results for the second workload (110 query and 20 random orderings). The second metric aims at specifying how much time each approach takes to define the query-execution schedule. Figure 2b presents the results for the first workload and Figure 3b for the second workload.

The effectiveness metric has been used to compare how many times each scheduling approach has presented a better result than FIFO approach (the queries are executed according to the random ordering), which was used as baseline. We have considered "Draw" when the absolute difference between execution times is less or equal than one minute (about 3 seconds of margin of error for each query instance). Figure 2c illustrates the effectiveness of the three proposed approaches for the first workload. IS and DRR presented better performance in 90% of cases, and GBA was better than FIFO in all cases. Regarding effectiveness for the second workload (Figure 3c), only IS approach was not better than FIFO.

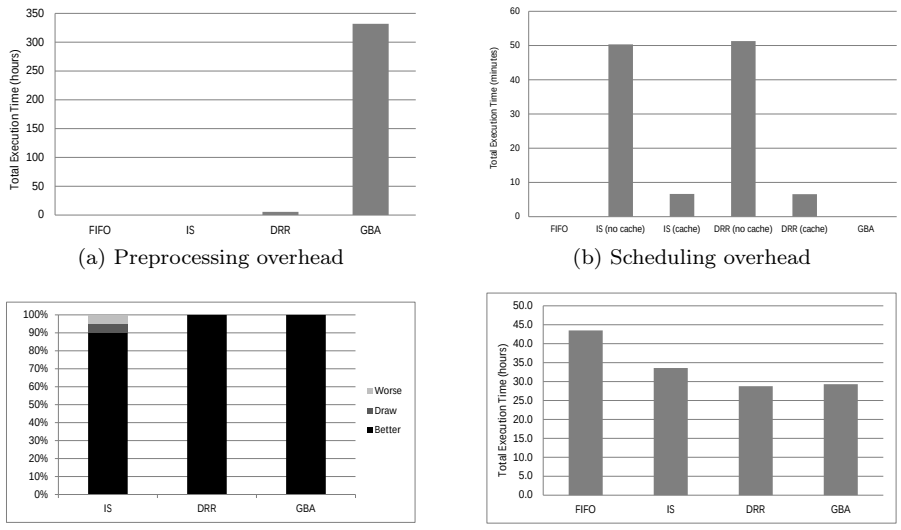
Efficiency has been applied to measure the total time consumed to execute queries by using the



(a) Preprocessing overhead (b) Scheduling overhead

(c) Effectiveness (d) Efficiency

Fig. 2: Comparative analysis between IS, DRR, GBA and FIFO with one instance per query type



(a) Preprocessing overhead (b) Scheduling overhead

(c) Effectiveness (d) Efficiency

Fig. 3: Comparative analysis between IS, DRR, GBA and FIFO with five instances per query type

schedules produced by the three approaches and by a FIFO schedule. As Figure 2d (results for the first workload) illustrates, IS was executed in 3.7 hours, saving 1 hour w.r.t. the execution time for the FIFO approach (22% of reduction). DRR had a better response time, since the schedule produced by DRR has consumed 3.5 hours to be executed (25% of reduction). Nevertheless, the most efficient strategy was GBA with reduction of 40%, saving 1.9 hours and hence being the only solution with execution time less than 3 hours.

Figure 3d brings the results of efficiency when the second workload (110 query and 20 random orderings) was executed. IS was executed in 33.6 hours, saving 9.9 hours w.r.t. the execution time for the FIFO approach (reduction of 23%). DRR had the best response time, during 28.8 hours (reduction of 33%). However GBA was also good with reduction of 33%, saving 14.7 hours.

The results presented in Figure 4 highlight the benefits of taking into account query interaction. For that experiment, we have picked up the execution time for seven TPC-H queries with highest

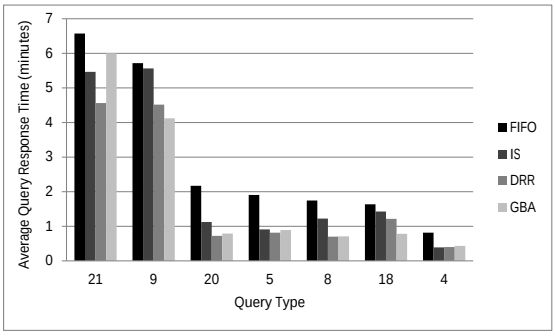


Fig. 4: Query type performance evaluation

execution time when execute in schedules produced by three approaches (IS, DRR, GBA) and FIFO. Observe that for each of those queries, the execution time is less than when they are executed in the FIFO approach. Recall that in FIFO query interaction is not considered.

6. CONCLUSION AND FUTURE WORK

In this article we propose three new approaches to model and measure query instance and query type interactions. Our approaches require no prior assumptions about the internal aspects of the database system or the reason of query interactions, making it non intrusive, namely, portable across systems. To demonstrate the profit of exploiting query interactions, we have developed a novel interaction-aware query scheduler for online workloads. The test results show that significant speed-ups are achieved. Besides, the evaluation shows that our approaches have potential to improve many database tuning algorithms. It may be interesting to incorporate the proposed approaches into other database system components such as query optimizer, physical design adviser and buffer manager, making them query-interaction aware. These possibilities represent interesting directions for future research.

REFERENCES

Ahmad, M., Aboulnaga, A., and Babu, S. Query Interactions in Database Workloads. In *Proceedings of the International Workshop on Testing Database Systems*. Rhode Island, USA, pp. 11:1–11:6, 2009.

Ahmad, M., Aboulnaga, A., Babu, S., and Munagala, K. Modeling and Exploiting Query Interactions in Database Systems. In *Proceedings of the ACM Conference on Information and Knowledge Management*. Napa Valley, California, USA, pp. 183–192, 2008.

Ahmad, M., Aboulnaga, A., Babu, S., and Munagala, K. Interaction-Aware Scheduling of Report-Generation Workloads. *The VLDB Journal* 20 (4): 589–615, 2011.

Chi, Y., Moon, H. J., Hacigümüş, H., and Tatemura, J. SLA-Tree: a Framework for Efficiently Supporting SLA-Based Decisions in Cloud Computing. In *Proceedings of the International Conference on Extending Database Technology*. Uppsala, Sweden, pp. 129–140, 2011.

Florescu, D. and Kossmann, D. Rethinking Cost and Performance of Database Systems. *SIGMOD Record* 38 (1): 43–48, 2009.

Niu, B., Martin, P., and Powley, W. Towards Autonomic Workload Management in DBMSs. *Journal of Database Management* 20 (3): 1–17, 2009.

Niu, B., Martin, P., Powley, W., Bird, P., and Horman, R. Adapting Mixed Workloads to Meet SLOs in Autonomic DBMSs. In *Proceedings of the IEEE International Conference on Data Engineering Workshop*. Istanbul, Turkey, pp. 478–484, 2007.

O'Gorman, K., Abbadi, A. E. E., and Agrawal, D. Multiple Query Optimization in Middleware Using Query Teamwork. *Software - Practice & Experience* 35 (4): 361–391, 2005.

Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. Efficient and Extensible Algorithms for Multi Query Optimization. *SIGMOD Record* 29 (2): 249–260, 2000.

Tan, K.-L. and Lu, H. Workload Scheduling for Multiple Query Processing. *Information Processing Letters* 55 (5): 251–257, 1995.

TPC. TPC-H Benchmark, 2013. `http://www.tpc.org/tpch/spec/tpch2.14.4.pdf`.