Sharing-Aware Scheduling of Web Services[*]

Junyan Jiang[1], Zhiyong Peng[1], Xiaoying Wu[2,**] and Nan Liang[1]

[1] Computer School, Wuhan University, China
{Jiangjy,peng}@whu.edu.cn, Ln9312@gmail.com
[2] State Key Laboratory of Software Engineering, Wuhan University, China
xiaoying.wu@whu.edu.cn

Abstract. The increasing and widespread use of web services, usually represented by database queries, is putting a strain on web database systems behind them. In such systems web services are associated with soft-deadlines, and the success of these systems (i.e., the user satisfaction) is better measured in terms of minimizing the deviation from the deadline, that is, tardiness. Previous work on query scheduling focused on ordering the execution of independent queries while ignoring the commonality among queries, such that a same work will be computed multiple times which can impact user satisfaction negatively. This paper proposes a new query scheduling framework which incorporates semantic caching techniques into the query scheduling procedure. We develop a query splitting-based strategy to discover common sub-expressions among queries and design a sharing-aware query scheduling algorithm **GASA** which minimizes average tardiness while reducing redundant work at the same time. We experimentally compare our approach with state-of-the-art methods on TPC-H workloads. Our experimental results show that our method can efficiently and effectively minimize average tardiness of a large number of data service requests.

Keywords: web services, query scheduling, semantic caching.

1 Introduction

The increasing and widespread use of web service is putting a strain on web databases systems behind them. These web services need to support SQL-style queries from form-based interface for strategic decision making in industries as varied as travel reservation, financial, insurance or even social networking. With the number of internet users and web services increasing, these systems are faced with loads that often involve hundreds or thousands of queries submitted at the same time [1]. In such highly interactive applications, user satisfaction or positive experience determines their success. Therefore it's crucial for such systems to prioritize execution of services as needed in order to keep users satisfied under varying workloads.

[*] This work is supported by National Natural Science Foundation of China under Grant No.61202035 and No. 61232002.
[**] Corresponding author.

One way to quantify a user's satisfaction is to associate a service/query with a soft-deadline which defines an upper bound (i.e. deadline) on the latency perceived by the end user accessing the results. The assigned deadline is a mapping from the service level agreements (SLAs) provided by the service provider to the end user. Hence, the success of the system (i.e., the user satisfaction) is better measured in terms of minimizing the deviation from the deadline, that is, tardiness.

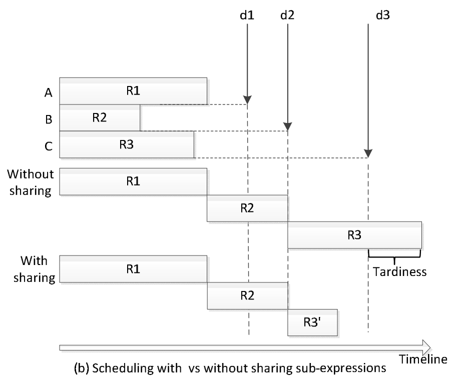
Previous work [4, 9, 10, 11, 12, 13] on query scheduling focused on ordering of the execution of independent queries while ignoring the commonality among queries. As a result, a same work will be computed multiple times which negatively impacts user satisfaction. Lots of techniques such as multi-query optimization [2] and semantic caching [3] can discover the query relevance by identifying commonality among queries and make use of intermediate results during the query execution to answer relevant queries. In this paper we propose a new query scheduling framework which takes into account of semantic caching techniques to improve user satisfaction. The intuition is that reducing evaluation cost of queries can help the minimization of tardiness of queries. Appropriately incorporating semantic caching into traditional scheduling methods brings us the following two challenges: First, we need a mechanism to model query relevance and develop a strategy to discover commonality among queries; second, we need to assess the impact of local query optimization on the global query scheduling. The following example illustrates the problem of traditional query scheduling method and the sharing opportunities among queries that can be exploited to minimize average tardiness.

```
Query A:
Select c_nationkey,c_mktsegment
From customer,orders,lineitem
Where o_orderdate < 1996-07-01'
and c_nationkey > 0 and c_nationkey < 20

Query B:
Select l_orderkey,o_orderdate,o_shippriority,
From customer,orders,lineitem
Where c_mktsegment = 'AUTOMOBILE'
and o_orderdate < '1995-03-26' and l_shipdate > '1995-05-26'

Query C:
Select c_nationkey,c_mktsegment
From customer,orders,lineitem
Where o_orderdate < 1995-07-01'
and c_nationkey > 10 and c_nationkey < 15
```

(a) queries A,B and C



(b) Scheduling with vs without sharing sub-expressions Under EDF strategies

Fig. 1. (a) Query Examples; (b) Scheduling Example

EXAMPLE 1 . Consider three queries A, B, C shown in figure 1(a), each query has a arrival time, a deadline, and a processing time. By *EDF* (Earliest Deadline First) strategy [16] we'll run query A first followed by B and C in that order. In this case, C will miss its deadline (see figure 1(b). However we can analyze the three queries and discover their commonalities. For example, the common sub-expressions between A and C are: "*o_orderdata<'1995-07-01*'" and "*c_nation*>10 and *c_nation* <15". We need to compute the common sub-expressions only once and cache their results.

The cached results can be used to evaluate C. This way, the evaluation cost of C is minimized and the tardiness of these three queries becomes 0.

Contributions. The main contributions of our paper are the followings:

- We propose a sharing-aware approach for query scheduling which exploits commonalities among queries.
- We develop a mechanism to model query relevance and design a strategy to discover commonalities among queries.
- We design a sharing-aware query scheduling algorithm **GASA** which minimizes average tardiness while reducing redundant work at the same time.
- We run extensive experiments to verify the efficiency and effectiveness of our proposed approach.
- To the best of our knowledge, our paper is the first one that combines classical query scheduling algorithms with semantic caching techniques.

Paper Organization. The rest of this paper is organized as follows: Section 2 presents related work. In Section 3, we describe preliminaries and our proposed problem. Section 4 illustrates our approach, and Section 5 presents the experiment. In Section 6 we discuss the conclusions.

2 Related Work

In this section, we introduce background information, including semantic caching and query scheduling. To our knowledge, no prior work considers both semantic caching and query scheduling in web databases system.

2.1 Semantic Caching

There are some related works on Semantic Caching. In a client-server system, a cache may be employed at the client-side in order to reduce the communication cost and improve query response time. A cache located at a client can only serve queries from the client itself, not from other clients. Such a cache is only beneficial for a query-intensive user. All techniques in this category adopt the dynamic caching approach. Semantic caching [3, 5] is a client-side caching model that associates cached results with valid ranges. Upon receiving a query Q, the relevant results in the cache are reported. A sub-query Q_0 is constructed from Q such that Q_0 covers the query region that cannot be answered by the cache. The sub-query Q_0 is then forwarded to the server in order to obtain the missing results of Q. Dar et al. [6] focuses on semantic caching of relational datasets. As an example, we assume that the dataset stores the age of each employee and that the cache contains the result of the query "find employees with age below 30." Now we assume that the client issues a query Q "find employees with age between 20 and 40." First, the employees with age between 20 and 30 can be obtained from the cache. Then, a sub-query Q_0 "find employees with age between 30 and 40" is submitted to the server for retrieving the remaining results.

The difference between our work and semantic cache is that we exploit intermediate results among queries while semantic cache reason queries based on pre-computed query results.

2.2 Query Scheduling

Scheduling is a well-studied research topic and is ubiquitous in many applications [7]. When per-query scheduling decisions are made where each query has a known execution time and deadlines, most problem instances become NP-complete [8]. Furthermore, the situation is not improved when the hard dead-lines are replaced by minimization of the number of deadline violations [9]. Therefore, most scheduling algorithms adopt certain heuristics. One family of such algorithms is cost-aware or value-based scheduling algorithms. In these algorithms, the decisions on scheduling are made so that certain costs are optimized. The costs could be defined in different ways: they could be fixed or time-varying values assigned to different queries [10, 11]; they could be about other metrics such as fairness [12] and result quality [13]. On minimizing average tardiness, [4] proposed an approach ASETS* efficiently minimizes query's tardiness integrating EDF and HDF/SRPT which highly relates our objective, the intuition is: at a given time t, we divide queries into two list, EDF list for those queries that can be finished before deadline and sorted by its deadline, and another list (SRPT) for queries that can't be finished before deadline sorted by its processing time. Each time we either (i) choose the first query from the EDF list or (ii) choose the first query from the SRPT list, the criterion to choose which query first is based on the impact of which order incurs less tardiness. However, none of these works considered sharing among queries during scheduling.

3 Preliminaries and Problem

In this section, we'll provide the query model and describe the system architecture. Finally we'll formally define our problem.

The query model is SQL queries associated with soft-deadline. We assume that a database D is given as a set of relations $\{R_1, R_2, \cdots, R_n\}$, each relation defined on a set of attributes. The expression of a SQL query only includes selection predicates which are called *tasks*. Given the query expression, we next define a partial order on tasks, including implication and satisfiability which characterize query relevance.

Definition 3.1(Query Relevance): We define query relevance in terms of *query predicate implication* and *query satisfiability*.

- *Query Implication*: A task ta_i implies task $ta_j (ta_i \Rightarrow ta_j)$ iff ta_i is a conjunction of selection predicates on attributes $A_1, A_2, A_3, \cdots, A_k$ of some relation R, and ta_j is a conjunction of selection predicates on the same relation R and on attributes $A_1, \cdots, A_l$ with $l \leq k$, and it is the case that for any instance of the relation R the result of evaluating ta_i is a subset of the result of evaluating ta_j.
- *Query Satisfiaiblity*: We call $ta_i \wedge ta_j$ is satisfiable, if that part of ta_i's answer is contained in ta_j.

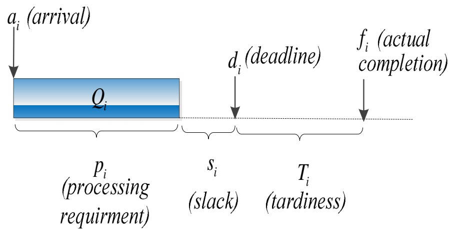


Fig. 2. Query Characteristics

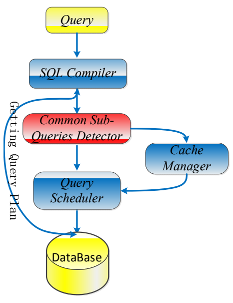


Fig. 3. System Architecture

Definition 3.2: We define the characteristics of a query to be (see Fig.2):

- Arrival Time (a_i): The time when Q_i has arrived at the database system.
- Deadline (d_i): The ideal time by which Q_i should finish execution.
- Processing Time (p_i): The processing time needed to execute Q_i. We assume that if caching or materialization is utilized for fragments, then Q_i's processing time is adjusted accordingly.
- Slack Time (s_i): The slack time of Query Q_i is the amount of time Q_i can take to finish after the deadline d_i. If the query cannot finish before the deadline then the slack time of query Q_i is 0.
- Tardiness ($T(Q_i)$): The tardiness of a query Q_i is the extra amount of time Q_i can take to finish after the deadline d_i. Specifically, at any beginning time t, $T(Q_i) = \max(f_i - d_i, 0)$, where $f_i = t + p_i$ is the actual completion time. Obviously, if the query is completed before the deadline, the tardiness of query Q_i is 0, otherwise the tardiness is larger than 0.

Given a set of queries, from the optimizer of the underlying database system, we can obtain the optimal plan of each query. From these plans, we can identify common sub-expressions among queries. Based on the common sub-expressions, a query can be split into two parts. One part is the common sub-expressions and the other is the non-common sub-expressions. Formally we provide a notion of query splitting as follows:

Definition 3.3(Query Splitting): Given two queries Q_i and Q_j, query splitting is the process of reducing the overlapped part of the two queries. Each query contains two parts: common sub-expression (we also call sub-queries) and non- common sub-expression. The overlapped part of queries (common sub-expression) is $C_{i,j}(Q_j, Q_j)$ (we may also use $C_{i,j}$ for simplicity) and the non-common sub-expressions of Q_i and Q_j are correspondingly $N_{i,j}$ and $N_{j,i}$. Note that if $C_{i,j}$ dose not exist, then $N_{i,j}$ equals to Q_i and $N_{j,i}$ equals to Q_j. The splitting strategies are presented in section 4.1. Once the query Q_i is split with Query Q_j, then the tardiness of the query Q_i is $T(Q_i) = \max\left(T\left(C_{i,j}\right), T(N_{i,j})\right)$

Definition 3.4(Query Processing Time Savings): Given two queries Q_i and Q_j, the query processing time saving is the extra time after the query splitting process.

Specifically we have$S_{i,j} = \mathrm{S}(Q_i, Q_j) = \max(0, p(Q_i) + p(Q_j) - p(C_{i,j}) - p(N_{i,j}) - p(N_{j,i}))$, evidently we only do those splitting which yield positive savings among queries while excluding negative ones.

3.1 System Architecture

The system architecture is shown in Fig.3. It consists of three modules including Common Sub-Expressions Detector, Cache Manager and Query Scheduler. The Common Sub-Queries Detector finds all the common sub-expressions among queries. The Cache Manger decides which query results to be cached into or be discarded from memory. The Query Scheduler utilizes several scheduling strategies to schedule queries to minimize the average tardiness. The three modules work on top of a database system.

3.2 Problem Statement

Next we define the query scheduling problem. Given a set of queries with the characteristics defined above where common sub-expressions can be identified among the queries, we find an order to execute the queries with the goal of minimizing the average tardiness while reducing redundant computations as much as possible. From the optimizer of the underlying database system, we can obtain the optimal plan of each query. From these plans, we can identify common sub-expressions among queries. We need a way to: 1) decide the execution order of queries; 2) select a subset of the common sub-expressions whose results need to be cached; 3) determine a subset of the queries which can reuse the cached results. The problem is formally defined as follows:

(Sharing-Aware Query Scheduling Problem) Given a query set $\bar{\mathrm{S}} = \{Q_i,\ 1 \leq \mathrm{i} \leq \mathrm{n}\}$ and their query splitting Strategies $C = \{C_{i,j}\}$. Then we want to minimize the average tardiness $\frac{1}{n}\sum_{\mathrm{i}=1}^{\mathrm{n}} T(Q_i)$ by ordering queries while reducing total query processing time $\sum_{i,j,i\neq j} S_{i,j}$ as much as possible by caching common sub-queries.

4 Sharing-Aware Scheduling Approach

In this section, we present in detail our approach which explicitly takes into account the sharing results among queries. At a high level, our approach makes use of the following insight: we consider the most important possible sharing opportunities among queries, together with their consequences in terms of average tardiness. First, we present our method to discover the relevance (i.e. the sub-expressions) among queries in section 4.1. Next we show our algorithm to schedule queries by reusing the common sub-expressions' results in section 4.2.

4.1 Discovering Common Sub-expressions among Queries

To discover common sub-expressions among queries, we need to determine whether a given pair of queries is relevant. We first use Table-Indicator to filter those queries which are definitely irrelevant, then we split queries based on their semantic relationship.

(a) Table-Indicator

Because most queries don't contain similar expressions, then we need to a fast filter with minimal overhead during the scheduling.

Definition 4.1: A table indicator TI_Q exists for a selection query Q iff Q represents SQL expression. If TI_s exists, it's a triple tuple TI_Q= [C_Q,S_Q,A_Q] where

- C_Q is the set of output columns in the selection clause of Q,
- S_Q is the set of source tables (or views) in Q,
- A_Q is the set of attributes in the where clause of Q.

Using the Table-Indicator, we can quickly detect the relevance of two queries $\mathrm{Q_i}$ and $\mathrm{Q_j}$ by check the following conditions: 1) $\mathrm{S_{Q_i}} = \mathrm{S_{Q_j}}$;2) $\mathrm{C_{Q_i}} \cap \mathrm{C_{Q_j}} \neq \emptyset$ and $\mathrm{A_{Q_i}} \cap \mathrm{A_{Q_j}} \neq \emptyset$. If $\mathrm{Q_i}$ and $\mathrm{Q_j}$ satisfy the above two conditions, then we can make them a group.

(b) Query Splitting

Detecting commonality among queries based on queries' table indicators, we have groups whose queries may share common sub-expressions. Then we compute the common predicates between each two queries using semantic caching method [5].

Suppose query Q_1's *task* $ta_{Q_1} = P_1 \wedge P_2 \cdots \wedge P_n$, and Q_2's *task* $ta_{Q_2} = P_1' \wedge P_2' \wedge \cdots \wedge P_m'$ (P_i is comparison predicate with the form of "attribute operator value"). Hence there are three types of relationships between Q_1 and Q_2(see Fig. 4).

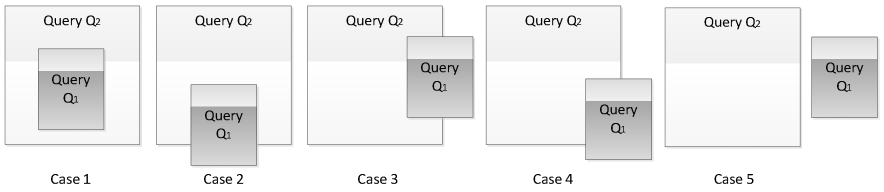


Fig. 4. Semantic relation between Query A and B

- The first type (case 1 in Fig.4) is query containment (i.e. implication): Q_2's result contain Q_1's or vice versa. Suppose Q_1 is contained by Q_2 (i.e. $ta_{Q_1} \Rightarrow ta_{Q_2}$, $\mathrm{C}_{Q_1} \subseteq \mathrm{C}_{Q_2}$), we have two splitting strategies: 1) we only evaluate Q_2 on database, and caching Q_2's result to evaluate Q_1; 2) we reformulate two queries to sub-expression $\mathrm{C}_{1,2}(Q_1, Q_2) = Q_1$ and $N_{1,2}(Q_1, Q_2) = Q_2 \wedge \neg Q_1$.
- The second type (case2-4 in Fig.4) is query overlap: we process query overlap based on the relationships of query's output attributes and the range predicate. Suppose Q_1 and Q_2 are overlapped, we have three types of overlapping

relationship:1) vertically overlap (case 2, $C_{Q_1} \subseteq C_{Q_2}$, $ta_{Q_1} \wedge ta_{Q_2}$is satisfiable); 2) horizontally overlap (case 3, $C_{Q_1} \cap C_{Q_2} \neq \emptyset$, $ta_{Q_1} \Rightarrow ta_{Q_2}$); 3) mixed overlap(case 3, $C_{Q_1} \cap C_{Q_2} \neq \emptyset$, $ta_{Q_1} \wedge ta_{Q_2}$is satisfiable). To reduce overlap between queries, we have three parts: $C_{1,2}(Q_1, Q_2) = Q_2 \wedge Q_1$, $N_{1,2} = \neg Q_2 \wedge Q_1$ and $N_{2,1} = Q_2 \wedge \neg Q_1$.

- The last one (case 5) is no connection between two queries: we have already processed this type using our table-indicators.

To facilitate our processing of detecting sub-queries, we have the following two heuristics to pruning the improper ones.

- ***Heuristics 1***(Containment size Check): Given two queries Q_1, Q_2, and $Q_1 \Rightarrow Q_2$, if $size(Q_1)/size(Q_2) < \alpha$ ($\alpha < 1$, here we take size of query as its result size in memory), then we will not cache Q_1's result, because sharing Q_1 doesn't greatly improve the total performance.
- ***Heuristics 2***(Exclude Sub-expressions With Huge Results): Given two queries Q_1 and Q_2, $C_{i,j}$ is their common sub-expression, if $\text{size}(C_{i,j}) > C_1$ (where C_1 is some constant), then we'll not cache $C_{i,j}$'s result, because it doesn't fit in the cache.

(c) Estimating Processing Time of Queries

An important aspect of ordering queries is to estimate queries' processing time which is orthogonal to our work, and there are two method we can use to estimate it by sampling the database [14] or by machine learning (ML) based method [15].The sampling method is to sample a small corpus from the original database as an alternative for the candidate queries. ML-based method is to learn the time from the training dataset. Considering that querying databases is time-consuming work, we adopt the sampling method [14] to estimate the query's processing time due to its efficiency.

4.2 GASA: *G*reedy *A*lgorithm for *S*haring-*A*ware Scheduling of Web Queries

Before we present the algorithm **GASA**, we have the following definitions.
The first list, EDF-List, contains all transactions that can still make their deadlines, if they start execution right now.

Definition 4.2: A query Q_i with deadline d_i is included in EDF-List iff, $\text{t} + r_i \leq d_i$, where t is the current time.

The second list, SRPT-List, contains all queries that already missed their deadlines.

Definition 4.3: A query Q_i with deadline d_i is included in SRPT-List iff, $\text{t} + r_i > d_i$, where t is the current time.

The main idea of the algorithm is to pick one query Q1 from the heads of the two lists and then choose another query Q2 from the remaining queries of the two lists such that the sharing of the sub-expressions' results of Q1 and Q2 can maximize the savings of the total query processing time for Q1 and Q2. We now describe the algorithm (The pseudo-code of Algorithm 1 is shown in figure 5) in detail as follows.

Algorithm 1. GASA($\bar{S}$,C)

Input: A set of queries with arrival time, estimated processing time and deadline.
Output: The id of the queries to run until next scheduling point and the id of common sub-expressions to cache.

```
1   Begin
2   for all newly arrived queries $Q_i$ do
3       Place $Q_i$ in the appropriate queue (EDF-List or SPRT-List)
4   end for
5   resort EDF-List & resort SRPT-List
6   while(EDF-List !=null and SRPT-List!=null)
7       $Q_{1,EDF}$ ←Top (EDF-List)
8       $Q_{1,SRPT}$ ←Top (SRPT-List)
9       if $r_{1,EDF} < (r_{1,SRPT} - s_{1,EDF})$ then $Q = Q_{1,EDF}$
10          else $Q = Q_{1,SRPT}$
11      end if
12
13      $S_Q = \{Q_j | Q_j \neq Q, Q_j$ and $Q$ is disjoint$\}$
14      j'=argmax$_{Q_j \in S_Q}(savings(\mathrm{Q}, \mathrm{Q_j}))$, )
15      Return $Q$, run $Q$ and cache $C(\mathrm{Q}, \mathrm{Q_{j'}})$'s result
16      refresh $\mathrm{Q_{j'}}$' expression and re-estimate processing time
17  end while
18  End
```

Fig. 5. GASA: *G*reedy *A*lgorithm for *S*haring-*A*ware Scheduling

Step 1: (line 7-11 in figure 5) We consider the impact of tardiness on the total set of queries, and compare the total tardiness of running $Q_{1,EDF}$. first and $Q_{1,SRPT}$ second with running $Q_{1,SRPT}$ first and $Q_{1,EDF}$ second, we have that if $\mathrm{r_{1,EDF}} < (r_{1,SRPT} - s_{1,EDF})$,then running $Q_{1,EDF}$ first will achieve lower tardiness, otherwise running $Q_{1,SRPT}$ will have a lower tardiness. By this checking, we then decide which query to run first.

Step 2: (line 12-14 in figure 5) To achieve sharing among queries, we need to decide which queries to share the picked queries' results. Suppose the picked query in the first step is Q, then we check which subset $\mathrm{S}(Q)$ of queries shares common sub-expressions with Q, and then we pick the query $Q_{j'}$ out of $S(Q)$ as the sharing query where the processing time saving of Q and $Q_{j'}$ is the maximal one, then we do splitting between Q and $Q_{j'}$, run query Q and cache their common sub-expressions' result.

Step 3: (line 15 in figure5) We refresh $Q_{j'}$ after splitting and estimate its processing time, then return to step 1 until no more queries exist.

GASA needs to be invoked in response to two types of events, the arrival and the completion of queries. We can use the standard balanced binary search tree as the priority queue, which requires only a time *O*(log *N*) to insert and update the priority lists. For splitting queries, we need *O(N)* to find common sub-expressions.

5 Experimental Evaluation

In this section, we describe our experimental settings and report our results.

5.1 Experimental Settings

Our experiments were conducted on the hardware configuration with 4-core 2.90GHz Intel CPU and 4GB memory running JVM 1.6.0 in Windows 7 Professional and data were stored in PostgreSQL 9.3.

We tested our method on TPC-H 1 GB databases. We chose 5 variant of TPC-H template queries without join predicates and aggregate predicates: #1,#4,#5 and # 6.We generated 250 queries, according to a Zipf distribution over the range[1-100] time units with the default Zipf parameter for skewness (α) set to 0.5 which was skewed for short queries. We chose 50,100,150,200 queries from the original 250 queries as a workload respectively. Arrival times of queries are assigned according to a Poisson process. The arrival rate of the Poisson distribution is set to be the rate of normal processed query number divided by the average query processing time. Each query is assigned a deadline $d_i = a_i + p_i + k_i * p_i$ where k_i is a factor that determines the ration between the initial slack time of a query and its processing time. k_i is generated uniformly over the range [0.0-k_{max}], where k_{max} is a simulation parameter with default value of 3.0.

We conduct the comparison in both the sharing and non-sharing cases. In the sharing case **GASA** is compared with ASETS*, EDF and SRPT [4], and also LS (least slack), under which the priority is $1/s_i$; in the non-sharing case, we compare **GASA** with EDF-Sharing, SRPT-Sharing LS-Sharing (which are EDF,SRPT and LS adapted with our greedy sharing strategies).

The performance of all the approaches is measured in terms of two metrics: (a) average tardiness which characterizes the total performance of our system, (b) total processing time savings of the whole workload.

5.2 Experimental Results

- **Comparison with Sharing-Nothing Polices**

In our first experiment, we measured the average tardiness for the four scheduling policies mentioned above as the number of queries increases from 0 to 250, with Zipf parameter $\alpha = 0.5$ and $k_{max} = 3.0$.

The experiment results for the average tardiness of the 4 scheduling policies on different query workload are shown in Figure 6. As we can see, when the number of queries is small, the system is able to meet most of the deadlines. In this case, the sharing opportunity for computing queries is small, however **GASA** still performs a little bit better than other policies. As the number of queries increases, the system cannot meet all the deadlines, whereas the sharing opportunity improves, and **GASA** substantially outperforms the other four polices. The maximum improvement by **GASA** is around 50.6% percent compared with ASETS* when the query number is 250.

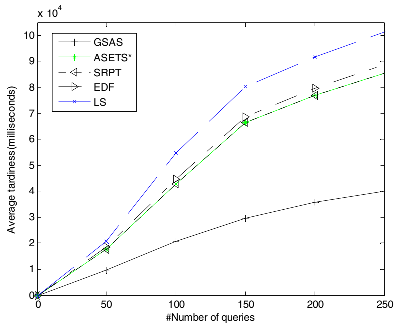
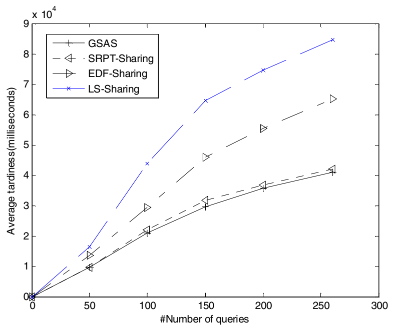


Fig. 6. Comparing GASA with state of art scheduling polices

- **Comparison with Baseline Sharing Polices**

We compare the performance of **GASA** with EDF-Sharing, SRPT-Sharing and LS-Sharing. Figure 7 shows the average tardiness comparison and figure 8 shows the average processing time savings.

In figure 7, **GASA** outperforms the other three polices in all the cases. LS-Sharing performs the worst, and the performance of SRPT-Sharing is comparable to **GASA** due to the high workload (in which most queries cannot meet their deadlines). In figure 8, the average processing time savings of all the approaches increase substantially when the number of queries increases from 0 to 50; when the number of queries is larger than 100, the savings begins to decrease. This can be explained by the fact that the rate of reduced processing time is smaller than the increasing rate of queries.



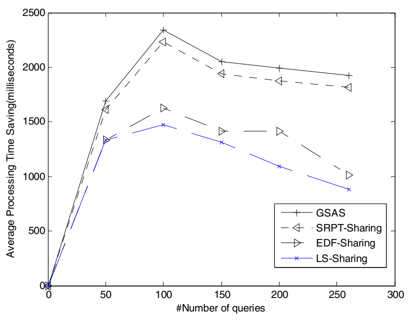


Fig. 7. Comparison of sharing-based polices **Fig. 8.** Average Processing Time Savings

6 Conclusions

In this paper we proposed the problem of sharing-aware scheduling of web services. We propose a sharing-aware approach for query scheduling which exploits commonalities among queries. We develop a mechanism to model query relevance and design a strategy to discover commonalities among queries. We design a sharing-aware query scheduling algorithm **GASA** which minimizes total tardiness while reducing redundant work at the same time. We run extensive experiments to verify the efficiency and effectiveness of our proposed approach. To the best of our knowledge, our paper is the first one that combines classical query scheduling algorithms with semantic caching techniques.

References

1. Unterbrunner, P.: Elastic, Reliable, and Robust Storage and Query Processing with Crescando /RB. PhD thesis, ETH Zurich (2012)
2. Zhou, J., Larson, P.-A., Freytag, J.C., Lehner, W.: Efficient Exploitation of Similar Subexpression for Query Processing. In: Proc. SIGMOD 2007, pp. 533–544 (2007)
3. Chidlovskii, B., Borghoff, U.M.: Semantic Caching of Web Queries. The VLDB Journal 9(1), 2–17 (2000)
4. Guirguis, S., Sharaf, M.A., Chrysanthis, P.K., Labrinidis, A., Pruhs, K.: Adaptive scheduling of web transactions. In: Proc. ICDE, pp. 357–368 (2009)
5. Ren, Q., Dunham, M.H., Kumar, V.: Semantic Caching and Query Processing. IEEE Transactions on Knowledge and Data Engineering 15(1), 192–210 (2003)
6. Dar, S.,Franklin, M.J., Þór Jónsson, B., Srivastava, D., Tan, M. : Semantic Data Caching and Replacement. In Proc. VLDB 1996, pp.330–341(1996).
7. Brucker, P.: Scheduling algorithms, 5th edn. Springer (2007)
8. Ullman, J.D.: Np-complete scheduling problems. J. Computer. Syst. Sci. 10(3), 384–393 (1975)
9. Peha, J.M.: Scheduling and dropping algorithms to support integrated services in packet-switched networks. PhD thesis, Stanford University (1991)
10. Haritsa, J.R., Carey, M.J., Livny, M.: Value-based scheduling in real-time database systems. In: Proc. VLDB 1993, pp. 117–152 (1993)
11. Irwin, D.E., Grit, L.E., Chase, J.S.: Balancing Risk and Reward in a Market-Based Task Service. In: Proc. 13th IEEE Int'l Symp. High Performance Distributed Computing, pp. 160–169 (2004)
12. Gupta, C., Mehta, A., Wang, S., Dayal, U.: Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. In: Proc. EDBT (2009)
13. He, Y., Elnikety, S., Larus, J., Yan, C.: Zeta: Scheduling interactive services with partial execution. In: Proc. SOCC 2012 (2012)
14. Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hakan, H., Naughton, J.F.: Predicting Query Execution Time:Are Optimizer Cost Models Really Unusable? In: Proc. ICDE 2013, pp. 1081–1092 (2013)
15. Malic, T., Rurns, R., Chawla, N.: A Black-Box Approach to Query Cardinality Estimation. In: Proc. CIDR 2007, pp. 56–67 (2007)
16. Schroeder, B., Harchol-Balter, M.: Web servers under overload: How scheduling can help. ACM Trans. Inter. Tech. 6(1), 20–52 (2006)