Revisiting Query Scheduling Policies for Lambda Functions

Karan Bavishi
kbavishi@wisc.edu

Abstract

Serverless computing has become an increasingly popular way of deploying applications because of reduced costs and deployment ease. Application developers need to only worry about the event-handling logic in 'functions', while the infrastructure provider takes care of the rest. Serverless computing reduces costs because developers only pay for CPU time when these functions are executing and not for idle capacity.

However, current query scheduling algorithms such as FIFO can result in higher costs for functions which execute only short queries. In this paper, we explore an alternative scheduling policy based on Earliest Deadline First, which also tries to exploit buffer pool locality by comparing predicates.

We show that such a policy can improve the median job turnover time by about 5.6×. But without the use of idle insertion time, it can result in about 2.7× worse turnover times in the 99th percentile.

1 Introduction

Ever since AWS launched its Lambda platform [3] in 2014, *serverless computing* has become a hot topic, promising easier deployment and reduced costs. Serverless computing refers to a new generation of platform-as-a-service offerings, such as Google Cloud Functions [7], Azure Functions [4] and IBM OpenWhisk [8], where code is deployed as 'functions' that operate as event handlers.

The infrastructure provider takes responsibility for most of the hosting and server logic, such as receiving client requests and responding to them, capacity planning, task scheduling and operational monitoring. The application developer only needs to worry about about the logic for processing client requests. Thus, instead of deploying continuously-running servers, we deploy 'functions' that operate as event handlers, and only *pay for CPU time when these functions are executing*.

This new pricing model of not having to pay for idle capacity has made its way to cloud DBaaS providers too. New DBaaS offerings, such as Google BigQuery [6], AWS Athena [2] and FaunaDB [5], have emerged which only charge based on the number of queries run and the amount of data scanned by those queries.

In this paper, we argue that commonly used query scheduling policies such as FIFO are unfair for short-lived lambda functions, and can result in higher costs. Researchers have shown that Shortest Job First (SJF) can be used to improve response time for shorter queries [19], but can cause starvation for long jobs.

We explore an alternative scheduling policy based on Earliest Deadline First (EDF), where we use the expected query completion time as the deadline. We also explore a buffer pool locality aware scheduling policy, which can exploit predicate similarity across queries to reduce the overall response time.

We show in Section 4.3 that non-preemptive EDF can improve median job turnover time by about 5.6×. However, the 99th percentile and the max turnover time in EDF is 4.3× and 4.2× worse than FIFO respectively. Using predicate similarity as a way to exploit buffer pool locality, we show that lo-

cality aware EDF improves 75th percentile and 99th percentile job turnover times by 8% and 1.6× over EDF.

The organization of this paper is as follows. In Section 2, we provide a deeper background of serverless computing and build motivation for why alternative scheduling policies are needed. In Section 3, we discuss the preliminary design of our locality-aware scheduler. Section 4 discusses the experimental methodology and our initial results. Section 7 describes the plan of this project before the submission deadline.

2 Background & Motivation

In this section, we discuss the rise of computing with an alternative pricing model where you only pay based on what you use, and how the current query scheduling policies are ill-suited for such computing models.

2.1 The rise of pay-per-actual-use computing

Lambda Functions - In 2014, Amazon Web Services (AWS) launched its Lambda platform [3] and this has spearheaded a new generation of platform-as-a-service offerings, such as Google Cloud Functions [7], Azure Functions [4] and IBM OpenWhisk [8]. These offerings are often referred to by the marketing term of "serverless computing".

Serverless computing refers to a new way of deploying services where the where the infrastructure provider takes responsibility for most of the service logic, such as receiving client requests and responding to them, capacity planning, task scheduling and operational monitoring. The application developer only needs to worry about about the logic for processing client requests. Thus, rather than continuously-running servers, we deploy functions that operate as event handlers, and only *pay for CPU time when these functions are executing*. Application developers are no longer responsible for managing the server process that listens to a TCP socket, hence the name 'serverless'.

This pricing model where you only pay based on your usage and not for idle time can drastically reduce the hosting and operating costs. For example,

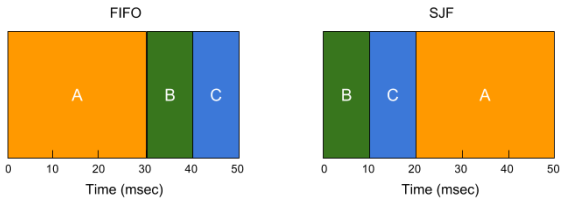

Figure 1: **Job turnover time for different scheduling policies**

Adzic et. al [11] describe a case study of a London-based social networking company, which reduced their hosting costs from $5000/month to less than $200/month, resulting in savings of more than 25×.

Going forward in this document, we will refer to these serverless offerings by the term *lambda functions* or *lambdas* for generality.

Cloud DBs with pay-per-query pricing - In recent years, database-as-a-service (DBaaS) offerings by cloud providers [1, 9] have become increasingly popular because of their lower operational costs as compared to hosting an RDMBS in-house, and their ability to scale-out on demand. The pricing model for these services generally involves having to pay a fixed price per hour based on the time the database is on.

Similar in spirit to the pricing model of lambda services of not having to pay for idle capacity, new DBaaS offerings have emerged which only charge based on the number of queries and the amount of data scanned. These offerings, such as Google BigQuery [6], AWS Athena [2] and FaunaDB [5], are better suited for lambdas and offer the potential to reduce costs significantly.

2.2 Unsuitability of current scheduling policies

Unfairness towards short-lived lambdas - Lambdas are charged based on the amount of time the function runs before it terminates. This means that a function will be charged even if it is sleeping waiting for a response from another service such as a database. It is known that traditionally used query scheduling policies such as FIFO hurt the response times for short queries [19].

FIFO can end up increasing the response times for short-lived lambdas with short queries, and thus

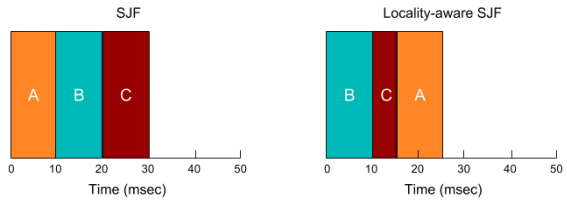

Figure 2: **Exploiting buffer locality in SJF for improving job turnover time**

increasing their execution time and the amount of money charged. To build motivation for this, we consider an example as shown in Figure 1. We have three queries A, B and C which run for 30 msec, 10 msec and 10 msec respectively. We consider two scheduling policies for a uniprocessor-like system, where the queries A, B and C arrived in that order. Using FIFO results in an average job turnover time of $\frac{30+40+50}{3} = 40msec$, whereas SJF results in an average turnover time of $\frac{10+20+50}{3} = 26.6msec$.

Exploiting buffer pool locality in query scheduling - Researchers have shown that query execution time can wary widely depending on the the number of pages already cached in the buffer pool [27]. DBaaS providers can exploit buffer pool locality across queries to reorder query scheduling and reduce overall turnover time.

To build motivation for this, we consider an example as shown in Figure 2. We have three queries A, B and C which are all expected to run for 10 msec each. Using SJF results in an average turnover time of $\frac{10+20+30}{3} = 20msec$. However, if query A and C had a similar predicate match for instance, we could reduce the execution time for C because some of the pages would already be cached in the buffer pool. Thus a locality-aware SJF scheduler would schedule C to run after A, and thus reduce the overall turnover time. For example, if running C after reduces its completion time to 5 msec as shown in Figure 2, we get an average turnover time of $\frac{10+15+25}{3} = 16.6msec$.

3 Preliminary scheduler design

We implement an external scheduler on top of the query fetch module in OLTPBench [16], which is used by worker threads to pull queries to be run. This external scheduler is similar to Gatekeeper [19]. It queues up incoming queries and schedules them using pre-established client connections to the RDBMS. We can support multiple query scheduling policies in our scheduler as described in the following subsection.

3.1 SJF Variant: Smallest Finish Time

Database query scheduling algorithms are often non-preemptive as most RDBMS do not support query pre-emption and resuming from the point of suspension. It is well known that using simple non-preemptive scheduling policies like FIFO can result in large response times for short queries [**?**]. Researchers have also shown that alternative policies such as Shortest Job First (SJF) can be used to improve the response times for short queries dramatically, but SJF can end up unfairly penalizing long jobs [19].

Preemptive EDF has been shown to be an optimal scheduling policy [23]. However, there are two main problems with using EDF. First, it requires accurate job deadlines or completion times. Second, it does not work for systems which do not support pre-emption such as databases. Therefore we use a non-preemptive version of Earliest Deadline First (EDF) in our system instead. Non-preemptive EDF has been shown to be non-optimal especially during high loads without the use of idle insertion times [22], but we leave such improvements to future work.

We use the estimated completion time of a query as its deadline. In other words, $T_{deadline} = T_{arrival} + Q_{runtime}$, where $Q_{runtime}$ represents the expected time for the query to complete. We use the query plan costs reported by PostgreSQL to estimate the query completion time. We assume a linear relationship between the query plan cost and query completion time for now. We use some offline measurements of the OLTPBench Twitter benchmark to derive the slope of the linear relationship.

It has been shown in prior work that using query plan costs leads to poor estimates of query execution time [21]. In theory, we can easily replace this with an online query execution time estimator like in [19] or use machine learning models [20] to

improve such estimates. We leave experimentation with more accurate completion time modules to future work.

3.2 Breaking down queries in lambdas

It may be beneficial to break down traditional join queries into independent queries to allow better query scheduling based on exploiting buffer pool locality.

```
SELECT * FROM tweets
WHERE uid IN (
    SELECT f2 FROM follows
    WHERE f1 = 42);
```

Listing 1: Query Version 1

```
SELECT f2 FROM follows
WHERE f1 = 42;
```

Listing 2: Query Version 2.1

```
SELECT * FROM tweets
WHERE uid IN (0,1,2,39,45);
```

Listing 3: Query Version 2.2

As an example, we demonstrate two versions of the `GetTweetsFromFollowing` job in Figure 3.2 and Figure 3.2, Figure 3.2 respectively. The first version (Version 1) uses a nested query which results in a join between the `tweets` and `follows` relations. The second breaks down the query into two parts (Version 2.1 and Version 2.2), where the second uses the results of the first. Expressing the query in this form allows a scheduler to reorder queries if there were previously run queries with similar predicates. We can exploit this to arrive at different cost estimates based on the similarity of the predicates.

3.3 Exploiting buffer pool locality via predicate comparison

Query execution times are known to vary wildly depending on the number of pages cached in the buffer pool [27]. Higher the number of pages cached, the lower the execution cost will be. We can use this effect to refine our cost estimates for queries, and thus ultimately optimize the query schedule.

As described in Section **??**, we test against a benchmark with one very commonly occurring query, namely the `GetTweetsFromFollowing` job. Our scheduler compares the predicates of a query of the form in Figure 3.2 with the predicates of previously run queries of the same form.

The cost of queries such as in Figure 3.2 largely depends on the number of predicates. In other words, the cost is of the form $Cost_q = A + n * B$, where n is the number of predicates and A and B are constants.

We incorporate matched predicates into this cost by reducing cost for each matched predicate. Thus the new cost formula is:

$$Cost_q = A + (n_{orig} - \frac{n_{match}}{discount\,factor}) * B$$

Here n_{orig} and n_{match} are the original number of predicates and the number of matched predicates respectively. We choose an initial value of 2 for the discount factor. Experimentation with other values for the discount factor is left as future work.

4 Preliminary Evaluation

4.1 OLTPBench Twitter

The Twitter benchmark in OLTPBench [16] is inspired by the popular micro-blogging website. It generates a synthetic dataset based on the characteristics of an anonymized snapshot of the Twitter social graph. It reflects important characteristics such as heavily skewed many-to-many "follow" relationships. We generate a data set consisting of 10 million tweets and 250,000 users, which translates to a database of size 3 GB.

The description and the frequency distribution of each query type in the workload is given in Table 1

4.2 Methodology

This section describes the methodology used for experimenting with different scheduling policies and using predicate matching for buffer locality.

Machine setup - We use a VM with 1 vCPU, 1 GB RAM and 10 GB HDD, running Linux 4.4.0-98 for our experiments. There is not enough RAM to hold the Twitter database (3 GB) in memory. We use a dedicated hypervisor node in CloudLab [28] to host our VM, and thus avoid interference from other

Query Type	Description	Freq
Get Tweet	Get tweet with given `tweet_id`	0.25%
Get Tweets From Following	Get tweets from followers for `uid`	91.00%
Get Followers	Get follower `uids` for given `uid`	0.25%
Get User Tweets	Get tweets for given `uid`	7.50%
Insert Tweet	Insert tweet for given `uid`	1.00%

Table 1: Twitter Query Mix

VMs. We set the CPU governor to performance in the hypervisor to disable the power save mode and downclocking.

RDBMS - We use PostgreSQL [10] version 9.5 as the RDBMS inside the VM. The main reason for this choice was that it was an open-source RDBMS which also provided query plan costs to the user. The latter is used in our non-preemptive EDF scheduling algorithm to estimate the query completion time. We configure PostgreSQL to use 256 MB for its shared buffer space, and keep default values for the other configuration settings.

Experimental methodology - We use OLTPBench to generate a query trace, and then reuse the same trace for all of our experiments to avoid making any incorrect conclusions. We run the benchmark with 10 client terminals. The chosen query arrival rate is 75 req/sec, which is close to the peak performance rate of the system (78.3 req/sec).

We run the Twitter benchmark for 3 different query scheduling algorithms: FIFO, EDF and predicate locality-aware EDF. For each experiment, we measure the job turnover time for each query that is run to completion. We run 10 experiments for each setting, and combine the measurements for all the 10 runs. Before starting a run, we restart PostgreSQL and dump all OS and filesystem caches.

4.3 Results

The CDF of query turnover times for FIFO, EDF and predicate locality-aware EDF is shown in Figure 3, and the percentile turnover times are listed in Table 2. The important observations are discussed below.

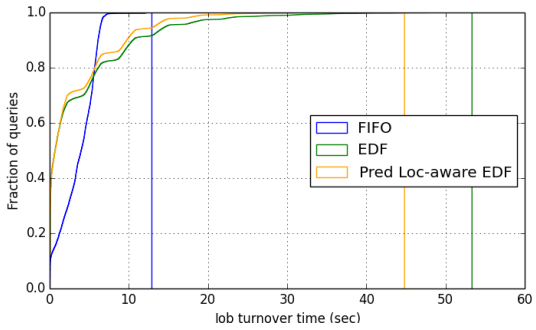

Figure 3: **Job turnover time CDF for different scheduling policies**

Policy	50%ile	75%ile	99%ile	Max
FIFO	3.99	5.49	7.07	12.91
EDF	0.71	5.28	30.19	53.34
Pred Loc-aware EDF	0.72	4.92	19.24	44.82

Table 2: Twitter Job turnover time (sec)

First, both EDF and predicate locality aware EDF improve median job turnover time by about 5.6×. This improvement shows that both algorithms improve turnover times for short queries as expected.

Second, using EDF results in significantly poor tail turnover times. The 99th percentile and the max turnover time in EDF is 4.3× and 4.2× worse than FIFO respectively. This is because we use a non-preemptive version of EDF which has been shown to penalize long jobs in overloaded systems without the use of idle insertion time [22]. Testing with an EDF version which uses idle insertion time like in [18] is future work.

Third, our approach of using predicate similarity to exploit buffer pool locality does seem to result in better query scheduling. Using predicate similarity improves 75th percentile and 99th percentile job turnover times by 8% and 1.6× over EDF. However this approach still suffers from the weaknesses of EDF of penalizing long jobs especially during high loads.

5 Deadline-aware scheduler

As seen in Section 4.3, Shortest Job First and its variant we tested Smallest Finish Time end up having significantly worse response times for anything beyond the 80th percentile. The main reason behind the poor performance of SJF is that it ignores deadlines associated with given queries. Most queries in Lambda Functions either have an implicit deadline due to the configured timeout for the Lambda, or have an explicit deadline specified via the timeout parameter specified using the query execution API

An optimal scheduler would be deadline-aware, drop queries during high loads and try to maximize the number of deadlines met. We look to research from the area of real-time scheduling with soft deadlines for inspiration to design this optimal query scheduler for Lambdas.

5.1 Group EDF (gEDF)

Researchers have found that Earliest Deadline First (EDF) is an optimal policy for preemptive scheduling scenarios. However, we need to consider non-preemptive scheduling policies because most databases do not support preempting running queries and then resuming them from suspension. Unfortunately, it has been proven that there is no such optimal policy for non-preemptive scheduling, and the problem of finding an optimal non-preemptive schedule is NP-hard. Previous efforts have found that using EDF in a non-preemptive setting can provide good results, but only in lightly loaded systems [**?**]. During heavy loads, EDF performs worse than simpler scheduling policies such as FIFO.

Instead, we focus on an alternative scheduling policy called Group EDF [**?**]. Group EDF (gEDF) is a hybrid between EDF and SJF. First, queries with similar deadlines are put in the same group. To schedule a query, we first select the group with the earliest deadline. Then we use SJF to break ties between queries within the same group. It has been shown that gEDF outperforms EDF in terms of the fraction of query deadlines met especially during high loads.

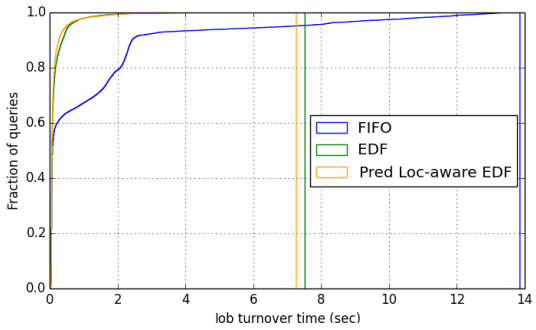

Figure 4: **Job turnover time CDF for different scheduling policies**

Policy	50%ile	75%ile	99%ile	Max
FIFO	0.09	1.72	12.06	13.86
EDF	0.08	0.14	1.59	7.53
Pred Loc-aware EDF	0.07	0.12	1.72	7.27

Table 3: Twitter Job turnover time (sec)

5.2 Evaluation: Lightly Loaded System

We first evaluate the performance of EDF in a lightly loaded system, and compare its performance to FIFO. We again run the OLTPBench Twitter benchmark as per the methodology described in Section **??**. We also extend EDF by adding predicate locality awareness and measure its impact. To simulate a lightly loaded system, we run the Twitter benchmark at roughly 50 transactions per second, which is close to half of the peak handling capacity of our PostgresSQL database.

EDFsettings - For both EDF and gEDF, we need to know the deadline of each input query. We configure $T_{deadline}$ for each query to be as follows:

$$T_{deadline} = T_{arrival} + D_{multiplier} * ExecTime$$

Here $T_{arrival}$ represents the arrival time of the query and $ExecTime$ represents the expected execution time for the given query. $D_{multipler}$ can be used to tune the strictness of the deadline query. A smaller value of $D_{multiplier}$ represents a stricter deadline. In our experiments below, we use $D_{multiplier} = 10$.

The CDF of query turnover times for FIFO, EDF and predicate locality-aware EDF is shown in Figure 4, and the percentile turnover times are listed in

Table 3. The important observations are discussed below.

First, using EDF can reduce the 75th percentile response time by 12×. This is similar to the improvements we saw with using SJF. Second, EDF does not suffer from bad tail performance as we saw with SJF in Section 4.3. This is because EDF drops queries if it doesn't expect to meet their deadlines. The dropping of queries occurs when the database is initially warming up, and the queries are taking longer to execute. Third, adding predicate locality awareness to EDF doesn't seem to help improve response times.

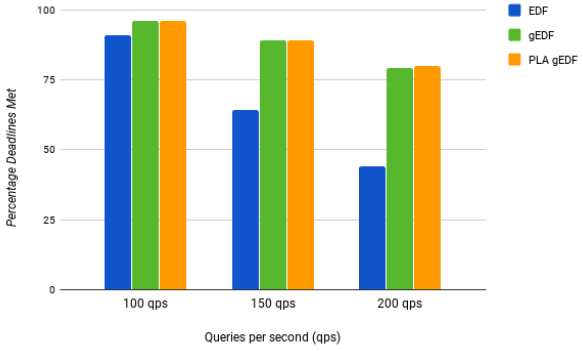

Figure 5: **Percentage deadlines met for different loads**

6 Evaluation: Heavily Loaded System

Non-preemptive EDF has been shown to perform well only in lightly loaded systems, and is known to perform badly during high loads [**?**, **?**]. Group EDF (gEDF) was shown to be able to meet more query deadlines than EDF especially during high loads [**?**]. In this section, we compare the performance of EDF with gEDF in a heavily loaded system. We measure the fraction of queries whose deadlines are met under different system loads.

gEDF settings - In gEDF, two queries Q_i and Q_j with deadlines D_i and D_j respectively belong to the same group if either of the two following conditions are true:

1. $D_i <= D_j <= (D_i + Gr * D_i)$, or
2. $D_j <= D_i <= (D_j + Gr * D_j)$

Gr is referred to as the group range parameter, and we select $Gr = 0.4$ similar to [**?**]. We continue to use $T_{deadline}$ and $D_{multiplier} = 10$ as described in Section 5.2.

Load Variation - We vary the load on the system by running the Twitter benchmark as specified in Section **??** at different input rates. We vary the input transaction rates from 100 transactions / sec to 200 transactions / sec. The former is very close to the peak capacity of the system and the latter represents twice the peak capacity.

The results for the comparison between EDF and gEDF is shown in Figure 5. We can see that as the load on the system increases to twice the peak capacity (200 qps), gEDF is able to meet 79% of its input query deadlines. On the other hand, EDF is only able to meet 44% of its deadlines. This shows that gEDF is very efficient at dropping queries whose deadlines it won't be able to satisfy.

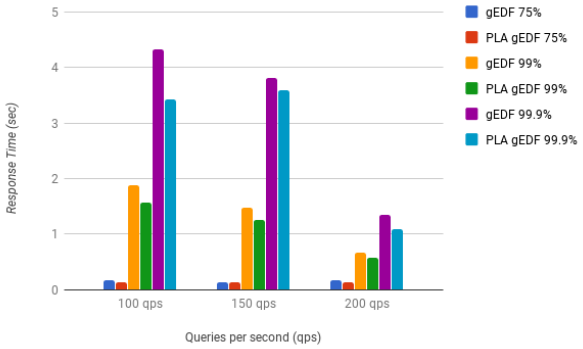

Figure 6: **Impact of locality awareness on response time percentiles: gEDF vs. predicate locality aware (PLA) gEDF**

We also evaluate the impact of predicate locality awareness by comparing the response time percentiles of gEDF and predicate locality aware gEDF (PLA gEDF). The results are shown in Figure 6.

Locality awareness helps improve tail response times. The 99th and 99.9th percentile times at the peak capacity (100 qps) for gEDF are lower than EDF by 18% and 21% respectively. Locality awareness seems to have slightly lesser impact at higher loads. At 200 queries per second, the 99th and 99.9th percentile times for gEDF are lower than EDF by 14% and 18% respectively.

7 Plan Ahead

In this section, we describe the plan going forward. The items are arranged in their decreasing order of priority:

- EDF with idle insertion - As was shown in Section 4.3, non-preemptive EDF can result in significantly worse turnover times for the 75th and 99th percentile in heavily loaded systems. Our plan is to explore an alternative policy such as clairvoyant EDF [18] which involves the use of idle insertion time.
- Online query execution time estimator - Currently, we use a static execution time estimator based on the cost plans reported by the optimizer. Alternatively, we could use an online estimator which revises the estimates based on runtime stats like in [19].
- Generic method for cost reduction due to buffer pool locality - Currently, we only demonstrated the potential of exploiting buffer pool locality by specifically looking at matching predicates. We aim to build a more generic cost matrix which computes reductions based on common indices, predicates and other things.

8 Related Work

Scheduling and admission control has been been studied before by database researchers. McWherter et al [26] study various lock and CPU scheduling policies for TPC-C and TPC-W workloads. We argue that our chosen benchmark, the Twitter benchmark in the OLTPBench suite [16], is more representative of modern web workloads than the TPC-W benchmark which has been marked obsolete since 2005. Elnikety et. al [19] use a transparent proxy called Gatekeeper which intercepts requests and provides admission control and request scheduling. Gatekeeper uses SJF to reduce response times for short queries, and uses an aging mechanism based on bounding the maximum service times for long queries. We argue that our EDF policy can work without over-penalizing long queries or needing any tuning for aging mechanisms. These works also ignore any potential gains due to exploiting buffer pool locality between queries.

Exploiting Query Interactions - Qshuffler [12, 13] uses statistical modeling techniques to capture both the positive and negative impact of query interactions, and use that to develop an interaction-aware query scheduler. However, using statistical modeling to understand interactions between OLTP queries may become in-feasible because of the potentially large number of query instances possible due to differing query parameters.

Our system tries to exploit locality across buffer pool pages by reordering queries based on the similarity of predicates of previously run queries. This idea of exploiting buffer pool locality is not new. Buffer-pool aware Query Optimization is studied in [27], where the authors explore the idea of using the number of cached pages to change query plan costs.

Our Earliest Deadline First (EDF) scheduling policy relies on getting reliable estimates for query completion times. There has been significant research in the area of estimating query progress and completion times [15, 24, 25, 21, 30]. Luo et. al [24] describe an approach to estimate progress for a single query, by breaking down the query plan into various non-blocking pipelines and using the query optimizer‘s cost estimates to determine progress. They extend this to a multi-query setting in [25], by accounting for resources assigned to multiple queries in the current mix.

We use the query plan cost as an estimate for the query completion time. Wu et. al [21] discuss the problems associated with using the query optimizer for estimating completion times, and suggest tuning the default RDBMS cost parameters. Other work [**?**, 14, 17, 20, 29] employs predictive frameworks based on statistical machine learning techniques for better execution time predictions. These techniques are complementary to our study, and can be easily integrated in our work to determine more accurate estimates and thus allow better query scheduling.

References

[1] Amazon RDS. https://aws.amazon.com/rds/.

[2] AWS Athena. https://aws.amazon.com/athena/.

[3] AWS Lambda. https://aws.amazon.com/lambda/.

[4] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[5] FaunaDB. https://fauna.com/serverless.

[6] Google BigQuery. https://cloud.google.com/bigquery/.

[7] Google Cloud Functions. https://cloud.google.com/functions/.

[8] IBM OpenWhisk. https://openwhisk.apache.org.

[9] Microsoft Azure SQL Database. https://azure.microsoft.com/en-us/services/sql-database/.

[10] PostgreSQL. http://www.postgresql.org.

[11] ADZIC, G., AND CHATLEY, R. Serverless computing: Economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 884–889.

[12] AHMAD, M., ABOULNAGA, A., BABU, S., AND MUNAGALA, K. Modeling and exploiting query interactions in database systems. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management* (New York, NY, USA, 2008), CIKM '08, ACM, pp. 183–192.

[13] AHMAD, M., ABOULNAGA, A., BABU, S., AND MUNAGALA, K. Interaction-aware scheduling of report-generation workloads. *The VLDB Journal 20*, 4 (Aug. 2011), 589–615.

[14] AKDERE, M., ETINTEMEL, U., RIONDATO, M., UPFAL, E., AND ZDONIK, S. B. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering* (April 2012), pp. 390–401.

[15] CHAUDHURI, S., KAUSHIK, R., AND RAMAMURTHY, R. When can we trust progress estimators for sql queries? In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2005), SIGMOD '05, ACM, pp. 575–586.

[16] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow. 7*, 4 (Dec. 2013), 277–288.

[17] DUGGAN, J., CETINTEMEL, U., PAPAEMMANOUIL, O., AND UPFAL, E. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 337–348.

[18] EKELIN, C. Clairvoyant non-preemptive edf scheduling. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)* (2006), pp. 7 pp.–32.

[19] ELNIKETY, S., NAHUM, E., TRACEY, J., AND ZWAENEPOEL, W. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA, 2004), WWW '04, ACM, pp. 276–286.

[20] GANAPATHI, A., KUNO, H., DAYAL, U., WIENER, J. L., FOX, A., JORDAN, M., AND PATTERSON, D. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proceedings of the 2009 IEEE International Conference on Data Engineering* (Washington, DC, USA, 2009), ICDE '09, IEEE Computer Society, pp. 592–603.

[21] HACIGUMUS, H., CHI, Y., WU, W., ZHU, S., TATEMURA, J., AND NAUGHTON, J. F. Predicting query execution time: Are optimizer cost models really unusable? In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)* (Washington, DC, USA, 2013), ICDE '13, IEEE Computer Society, pp. 1081–1092.

[22] JEFFAY, K., STANAT, D. F., AND MARTEL, C. U. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium* (Dec 1991), pp. 129–139.

[23] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20*, 1 (Jan. 1973), 46–61.

[24] LUO, G., NAUGHTON, J. F., ELLMANN, C. J., AND WATZKE, M. W. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2004), SIGMOD '04, ACM, pp. 791–802.

[25] LUO, G., NAUGHTON, J. F., AND YU, P. S. *Multi-query SQL Progress Indicators*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 921–941.

[26] MCWHERTER, D. T., SCHROEDER, B., AILAMAKI, A., AND HARCHOL-BALTER, M. Priority mechanisms for oltp and transactional web applications. In *Proceedings. 20th International Conference on Data Engineering* (March 2004), pp. 535–546.

[27] RAMAMURTHY, R., AND DEWITT, D. J. Buffer-pool aware query optimization. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings* (2005), pp. 250–261.

[28] RICCI, R., EIDE, E., AND THE CLOUDLAB TEAM. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login: 39*, 6 (Dec. 2014).

[29] TOZER, S., BRECHT, T., AND ABOULNAGA, A. Q-cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (March 2010), pp. 397–408.

[30] WU, W., CHI, Y., HACÍGÜMÜŞ, H., AND NAUGHTON, J. F. Towards predicting query execution time for concurrent and dynamic database workloads. *Proc. VLDB Endow. 6*, 10 (Aug. 2013), 925–936.