Dynamic Query Prioritization for In-Memory Databases

Johannes Wust[1], Martin Grund[2], Hasso Plattner[1]

[1]Hassso Plattner Institute, D-14440 Potsdam
[2]University of Fribourg, CH-1700 Fribourg

johannes.wust@hpi.uni-potsdam.de, grund@exascale.info
hasso.plattner@hpi.uni-potsdam.de

Abstract: In-memory database management systems have the potential to reduce the execution time of complex operational analytical queries to the order of seconds while executing business transactions in parallel. The main reasons for this increase of performance are massive intra-query parallelism on many-core CPUs and primary data storage in main memory instead of disks or SSDs. However, database management systems in enterprise scenarios typically run a mix of different applications and users, of varying importance, concurrently. As an example, interactive applications have a much higher response-time objective compared to periodic jobs producing daily reports and should be run with priority. In addition to strict prioritization, enforcing a fair share of database resources is desirable, if several users work on applications that share a database. Solutions for resource management based on priorities have been proposed for disk-based database management systems. They typically rely on multiplexing threads on a number of processing units, which is unfavorable for in-memory databases on multi-cores, as single queries are executed in parallel and numerous context switches disrupt cache-conscious algorithms. Consequently, we propose an approach towards resource management based on a task-based query execution that avoids thread multiplexing. The basic idea is to calculate the allowed share of execution time for each user based on the priorities of all users and adjust priorities of tasks of incoming queries to converge to this share.

1 Introduction

In-memory database management systems (IMDBMS) that leverage column-oriented storage have been proposed to run analytical queries directly on the transactional database schema [Pla09]. This enables building analytical capabilities on top of the transactional system, leading to reduced system complexity and reduced overall operating cost. However, running multiple, potentially different applications on a single database instance that records business events leads to a mix of heterogeneous queries that may have different response time objectives.

With TAMEX [WGP13], we have proposed a task-based framework for multiple query class execution on IMDBMS. TAMEX allows to statically prioritize classes of workloads, for example transactional queries over analytical queries to achieve almost constant response-time of transactions independent of the analytical workload. However, static pri-

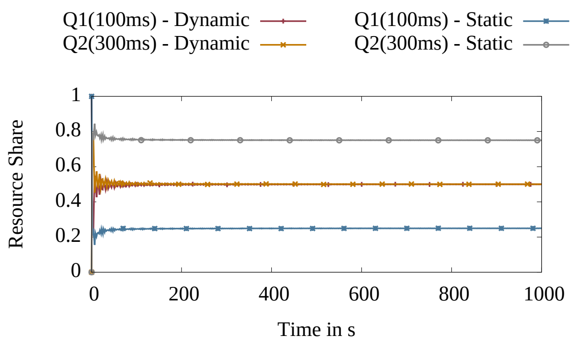


Figure 1: Comparing static and dynamic query priorities for a single priority class

oritization falls short on enforcing a fair share of database resources among sessions with different query execution times, since users with similar priorities are scheduled strictly first-in-first-out, independent of the execution time. Enforcing a fair share is desirable in many scenarios, where many users work concurrently on a shared database system.

We further illustrate the shortcoming of static priority-based scheduling using a simple example: Assuming that two concurrent sessions are connected to the database and simultaneously issue queries. As both sessions are connected as analytical clients they will be assigned the same static priority for executing their queries. Now, the first session issues queries that are executed in 100ms and the second session issues queries against the database that take on average 300ms. If all queries are executed without think time and sequentially, for simplicity we assume a single processing unit, they will basically interleave. As long as the difference in query execution time between these two sessions is not too big, this will not result in any performance degradations. However, in the above case the average response time of the query will be dominated by the wait latency for the longer query and quickly approach 400ms. For the heavier query the additional latency does not have as big an impact and it will account for close to 75% of the consumed resources.

We analyzed this motivating use-case with a scheduling simulator to compare dynamic and static query priorities. Figure 1 shows the result of this simulation. As expected, in the case of the static priorities the second longer query consumes the majority of the resources thus violating the fair- share scheduling. Using dynamic priorities as proposed in detail in this paper, the scheduler will distribute the available resources equally among the two queries independent of the actual run-time of the query.

To summarize the above simulation, we can derive that traditional queue-based scheduling for fair-share scheduling works only well for such scenarios, where the independent time quanta that are executed, are roughly equal or the tasks can be preempted. Both properties do not hold true for task-based scheduling in in-memory databases as tasks can have varying sizes and can be typically not be preempted.

In this paper, we propose an extension of TAMEX that enforces a fair share of database execution time by dynamically adjusting priorities of queries. The remainder of the paper is structured as follows: In the next section, we give a brief overview of the assumed system model and in Section 2.2 the task-based query execution with TAMEX. Section 3 describes our model for dynamic query prioritization and Section 4 the architecture of our extension to TAMEX. In Section 5, we evaluate our proposed solution with an enterprise typical query workload. The next section discusses related work and the last section closes with some concluding remarks and directions for future work.

2 System Model and Task-based Query Execution

This section gives a brief overview of the underlying system model of an IMDBMS, as well as the task-based query execution framework TAMEX [WGP13], which we use for implementing dynamic query prioritization.

2.1 System Model

We assume an IMDBMS following the system model described in [Pla11], where data is physically stored decomposed in a column-oriented structure. To achieve high read and write performance, an insert-only approach is applied and the data store is split in two parts, a read optimized main partition and a write optimized differential store [KKG$^+$11]. We apply a multi version concurrency control (MVCC) based on transaction IDs (TID) to determine which records are visible to each transaction when multiple transactions run in parallel. See [Pla11] for more details. As our proposed approach for dynamic query prioritization is largely agnostic to specific architectural details of the database. it can be easily generalized and applied to other architecture. However, our approach assumes that the execution of queries can be split in small atomic tasks, which can be executed in parallel, as we will explain in the next section.

2.2 Task-based Query Execution with TAMEX

This section gives an overview of the task-based query execution framework TAMEX, which is implemented based on HYRISE [GKP$^+$10].

We understand task-based query execution as the transformation of the logical query plan into a set of atomic tasks that represent this plan. These tasks may have data dependencies, but otherwise can be executed independently. We consider such an atomic task as the unit for scheduling. Compared to scheduling whole queries, a task-based approach provides two main advantages: better load balancing on a multiprocessor system, as well as more control over progress of query execution based on priorities. The second advantage is achieved as splitting queries into small units of work introduces natural scheduling intervals during query execution, where lower priority queries can be paused to run higher priority queries without the need of canceling or preempting the low priority query. Assuming a sufficiently small task size, processing units can be freed quickly to execute incoming high priority queries. With the advent of modern many-core processors, the efficient splitting of monolithic queries becomes more and more important as for example stated in [BTA13].

TAMEX adopts this concept by transforming incoming queries into a directed acyclic graph of tasks and schedules these tasks based on priorities. For TAMEX, we extended HYRISE to support parallel execution of queries, as well as intra-query parallelism, based on multi-threading. Figure 2 provides an overview of the main components of TAMEX and the extensions for dynamic query execution, which are explained later in Section 4.2; a more detailed description of TAMEX is provided in [WGP13]. An incoming query is compiled and transformed into a task graph. The task scheduler assigns all ready tasks to a priority queue; all tasks with unmet dependencies are placed into a wait set until they become ready. Worker threads of a threadpool take the tasks from the queue and execute them. Each thread is assigned to a physical processing unit and executes one and only one task. That way, incoming high priority tasks can start immediately executing on all processing units, once the currently running tasks have finished. While this static scheduling approach can effectively prioritize a query class over another, it cannot enforce a fair share of resources if queries with similar priorities are issued. In this paper, we build on TAMEX by setting these priorities dynamically to enforce a given resource share for query classes.

3 Dynamic Shared Query Execution

As motivated in the Introduction, fair resource sharing is of great importance in systems with heterogeneous workloads. In this section, we will introduce the concept of *Dynamic Shared Query Execution* with the goal to approximate a fair resource usage between database sessions on a single system.

In the following, we will describe a new dynamic shared query scheduler with the objective of scheduling queries from independent session on a fair distribution of the available computing hardware. We achieve a good scheduling performance by dynamically re-calculating priorities of the different queries of independent sessions so that resources consumption is better distributed. Since scheduling of queries is a time-critical operation we take special care in optimizing these operations to minimize the impact of dynamically adjusting the priorities. In addition, it is possible to manually decide whether or not dynamic priority adjustments should be made available for the different priority classes. As a result, we maintain high throughput for transactional queries without additional scheduling overhead, but achieve a better flexibility for medium and long running queries.

3.1 Work-share Definition

We consider a database management system running on a server with N processing units and S open database sessions during an interval T. Each session $s_i \in S$ has an assigned priority p_i and a set of executed queries $Q_i(t)$ at any point in time t during T. Each time a query s finished, it is added to Q_i. We consider online arrival of queries, meaning that the database has no knowledge about the future arrival of queries. Each query $q_{i,j} \in Q_i$ is defined by a set of tasks $O_{i,j}$ and an arrival time $t_{i,j}$. Each task $o_{i,j,n}$ is executed sequentially on one processing unit $n_i \in N$ and has an assigned amount of work $w_{i,j,n}$

processed by the database. In our model, a task has exclusive access to a single processing unit and cannot be preempted.

For each session s_i we determine the of work w_i that the database has been executed on behalf of this session at a time t, by

$$w_i(t) = \sum_{q_{i,j} \in Q_i(t)} \sum_{o_{i,j,n} \in O_{i,j}} w_{i,j,n} \tag{1}$$

and the total work W processed by the database by

$$W(t) = \sum_{s_i \in S} w_i(t) \tag{2}$$

The share of work ws_i of a session s_i at time t is calculated by

$$ws_i(t) = \frac{w_i(t)}{W(t)} \tag{3}$$

Based on the provided priorities p_i for each session, each query has a target share ts_i, defined by

$$ts_i = \frac{p_i}{\sum_{s_j \in S} p_j} \tag{4}$$

We define the relative share deviation of ws_i from ts_i as

$$\Delta s_i(t) = \frac{ts_i - ws_i(t)}{ts_i} \tag{5}$$

Based on the provided definition, we can formulate the problem of shared query execution as:

Definition 1 *Let $S = \{s_1, ..., s_n\}$ be the set of active sessions in an interval T with priorities p_i and queries Q_i, executed on a database with N processing nodes. The problem to solve is to provide an assignment of processing units to tasks $o_{i,j,n}$ during T that minimizes the overall deviation of the work share from the target share over an interval T:*

$$\Delta S = \int_0^T \sum_{s_i \in S} |ts_i - ws_i(t)| \tag{6}$$

Due to the online arrival of queries, a scheduling algorithm that assigns processing nodes to tasks of queries cannot guarantee optimal schedules. As we assume non-preemptiveness of tasks, it is possible to find examples for which an online algorithm produces results far from optimal [LKA04]. A competitive-analysis or worst-case analysis will produce largely meaningless results. Therefore, we provide a heuristic approach and experimentally validate its effectiveness.

4 Architecture

This section introduces our heuristic approach for approximating the problem described in Section 3 and a brief overview of the implementation.

4.1 Approximation of Shared Query Scheduling

The basic idea of our approach is to measure the actual work spent on of query processing for each session and calculate the relative share deviation $\Delta s_i(t)$ for each session $s_i(t)$ between certain points in time t. Based on the ranking of the relative share deviation, we assign priorities to queries with the objective of minimizing the relative share deviation.

To approximate the overall work share deviation for each user, we have implemented moving average and exponential smoothing [Bro04], both first and second order, as heuristics. As we found it hard to justify the choice of parameters for exponential smoothing and as we obtained more predictable results with the moving average, we limit our discussion here on the moving average. To calculate the work shares, we accumulate the work processed for each user, after a task has been completed. In fixed time intervals, we calculate the work share defined in Equation 3. For the moving average, we take the average work share over the last n intervals to calculate the average work share deviation of Equation 5:

$$ws_i(t) = \frac{1}{n} \sum_{\{t-n,\ldots,t\}} \frac{w_i(t)}{W(t)} \tag{7}$$

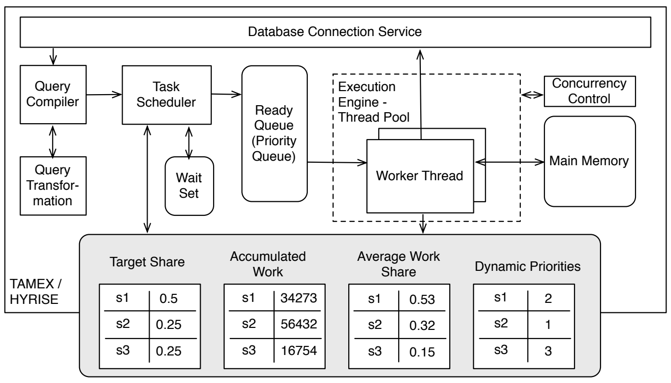
In Equation 7, $w_i(t)$ defines the accumulated work of session i over the last observed period. To assign the dynamic priorities to the session, we use the work share deviation to sort the sessions and map the priorities accordingly. This approach introduces two parameters that can be modified to adjust the scheduler to the current workload. The first parameter is the window size n of the moving average, as it defines the impact of the currently observed workload compared to the past, and the second parameter is the interval that is used to evaluate a possible change in priorities.

4.2 Architecture for Shared Query Scheduling

We have implemented our approach to approximatively solve the dynamic shared query execution problem described in Section 3 based on our database storage engine HYRISE [GKP$^+$10] and our task-based execution framework TAMEX [WGP13], introduced in Section 2.2.

Figure 2 shows an overview of the extension to TAMEX. For each session, we keep track of the target share calculated by Equation 4, the work processed for each session in the current time interval (indicated as Accumulated Work), the average work share and the dynamic priorities. After a task is completed, the execution time of this task is added to the accumulated work for the corresponding session. At the end of an interval, an update process calculates the relative work share deviation and assigns the dynamic priorities accordingly to minimize the deviation in the next interval.

The update process consists of the following steps: we calculate the work share as defined in Equation 3 by dividing the accumulated work for a session by the total work of all sessions during the considered interval. Once read, the accumulated work is reset. Next, we incrementally calculate the average work share using Equation 7 and determine the relative work share deviation for each user using Equation 5. As a last step, we sort all sessions in descending order by this deviation and assign dynamic priorities accordingly, giving the



Target Share	
s1	0.5
s2	0.25
s3	0.25

Accumulated Work	
s1	34273
s2	56432
s3	16754

Average Work Share	
s1	0.53
s2	0.32
s3	0.15

Dynamic Priorities	
s1	2
s2	1
s3	3

Figure 2: Dynamic query prioritization based on TAMEX

highest priority to the session with the highest relative work share deviation. It is important to mention, that the worker threads executing tasks are not disrupted by the update process. Figure 2 illustrates the recorded data and the resulting dynamic priorities. Session $s3$ gets the highest priority as it has the largest work share deviation. If the task scheduler places a new task, or one from the *Wait Set* that becomes ready, in the *ReadyQueue*, it updates the priority of the task according to the dynamic priority of the session.

To achieve the highest possible accuracy the task scheduler would have to provide global state information about the actual work of each session that is then updated by the individual execution threads as soon as a single task is finished. A drawback of this global work share calculation is the global dependency to accumulate the total work. To alleviate this dependency, we use an atomic hash-map that maps the individual sessions to a local counter value. Now, this state is not shared among all execution threads, but only the threads working on tasks of the same session access a unique storage location.

This situation can be additionally improved by keeping a copy of this session map in the thread-local storage of each execution thread that is only lazily collected from the scheduler once it detects a recalculation of the priorities for the tasks. Using the thread-local approach basically avoids contention for the session based work share completely as all manipulations are performed thread-local and only a single consumer will retrieve the individual items.

The adjustment of the dynamic priorities is triggered by the worker threads notifying the task scheduler when a task is finished. If the time interval for calculating an average work share has been passed, the update process, as described above, is initiated. As we need to sort the list of sessions by relative share deviation, the complexity is $O(nlogn)$, with n being the number of sessions. In practice we have compared the performance of TAMEX with and without our extension and could not determine significant performance penalty for up to a 1000 concurrent users.

Since user sessions can be inactive during a period of time when we reevaluate priorities, we only consider those sessions that have issued work over this period of time. As long as the session is inactive, it will not bias the priority calculation; when the session is reactivated, we start the calculation of the moving average again, without considering the share prior to the inactivity.

5 Evaluation

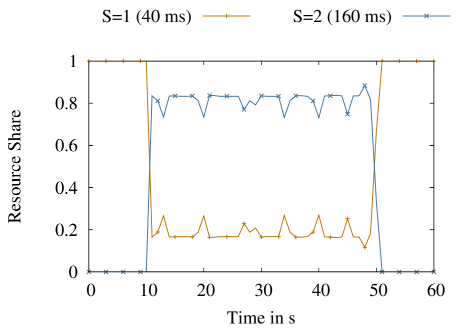
This section provides an experimental evaluation of our approach towards dynamic query prioritization, which we described in Section 4. Our test machine is equipped with 2 Intel(R) 5670 CPUs with 6 cores each and 144GB RAM. The first two experiments demonstrate the effectiveness of our approach to dynamically adjust priorities to converge to a desired target share. In the third experiment, we evaluate parameters for calculating the moving average and derive recommendations for choosing them appropriately.

Motivated by the introductory experiment illustrated in Figure 1, we have set up an experiment with two sessions, each consisting of a number of equivalent users that issue a stream of queries to the database without think time. Each query consists of two table scans and a join, whereas each operator runs in parallel up to a degree of intra-operator parallelism of 12, corresponding to the number of threads running in parallel. Due to a different size of input tables, the query issued by the users of session 1 (S=1) takes 40ms processing time in the database kernel and the query of session 2 (S=2) 160ms. Each query has 154 tasks, with a maximum task runtime of about 50ms for the longer query. We ran the experiment with these two sessions using a round robin scheduler, as well as our fair share scheduler that enforces an equal resource share for both sessions. Each time, the experiment ran for 60s, whereas the second session started after 10s and ended after 50s. We have chosen the window size n of Equation 7 to be 50 and the interval for updating priorities to 0.2s.

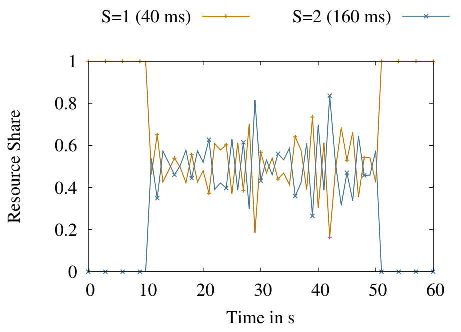
Figure 3(a) shows the result for the round robin scheduler. For each second, we have plotted the resource share of the last second. As we take the point of view of a user outside of the database, we count work processed for a session at the point of time when an entire query is finished, as opposed to single tasks. In line with our expectations from the simulation, applying a round robin scheduler leads to a share equal to the ratio of the runtime of both queries. In Figure 3(b), we see that the dynamic prioritization of queries leads to a varying resource share of each queries averaging to a fair share over the interval between 10 and 50s. While the round robin fails to distribute the resources equally among the two sessions, it becomes possible to efficiently schedule queries with different runtimes and to distribute the resources equally when applying dynamic query prioritization.

Figure 4 demonstrates the applicability of our approach to a larger number of sessions and different priorities. This time, all sessions S consist of a single user issuing a stream of the query described above with 160ms processing time when executed as a single query on the system. When scheduling all incoming tasks with a round robin scheduler, each query gets approximately the same share of the system resources (Figure 4(a)). In Figure 4(b), we assigned User 1 a priority of 4 (P=4) and the remaining users a priority of 1 (P=1) with the objective of enforcing a share of 50% of the total resources for User 1 and 12.5% for each of the other users during the interval when all users issue queries in parallel. In this experiment, our dynamic query prioritization is able to schedule the queries of all the different sessions according to the assigned priorities.

Choosing the window size for the moving average and the interval length of updating

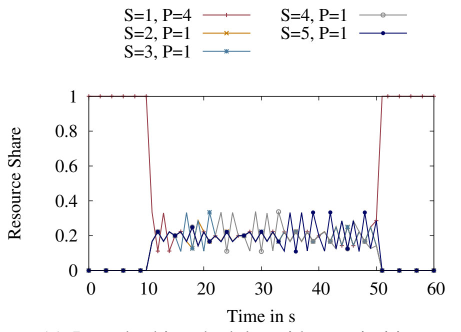


(a) Round robin scheduler wihout priorities

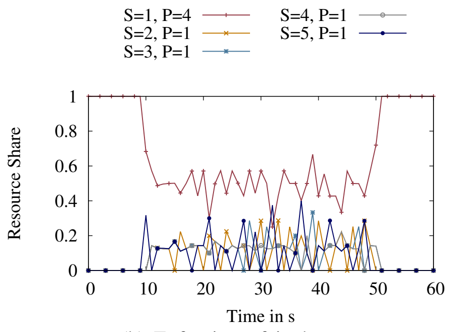


(b) Enforcing a fair share

Figure 3: Two sessions issuing queries with different execution times



(a) Round robin scheduler without priorities



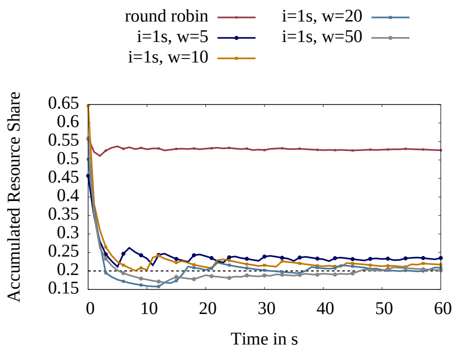
(b) Enforcing a fair share

Figure 4: Five sessions issuing queries (160ms) with different priorities

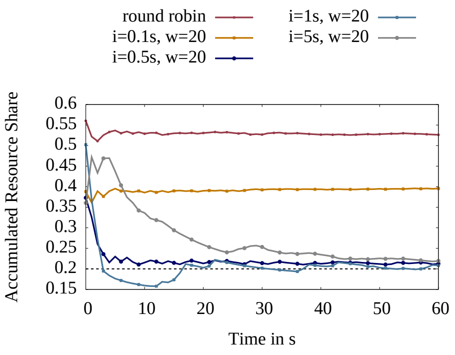
priories is a trade-off between overall accuracy and adaptation time to react on changes in the workload. To illustrate this, we have tested five sessions with equal priorities, issuing a constant stream of queries. One session issues a query with 160ms runtime, the other users a query with 40ms run time. We start all users at the same time and measure the cumulated work share since the start for 60s. Figure 5 shows the results for the calculation of the relative share deviation with moving average for different window sizes (w) and interval lengths (i) for one of the five sessions with query length 160ms.

In Figure 5(a), we have changed the window size for the moving average and kept the size of the observation interval constant at 1s. As expected, a larger window size leads to a smoother curve that converges to the target share of 20% without major deviations. A smaller window size shows more spikes, as intervals with above or below average have a larger impact on calculated work share, but also adapts faster to workload changes. However, if the window size is chosen too small, as it is here the case for size 5, the scheduler cannot enforce the overall target share anymore, as the sample size is too small.

In Figure 5(b), we changed the interval lengths for the moving average and kept the window size constantly at 20. For small interval lengths of 0.1s, the total time interval of window size multiplied with interval lengths that is considered becomes so small, that the scheduler cannot systematically decrease the performance of the user with the long run-



(a) Changing the window size



(b) Changing the interval length

Figure 5: Comparing parameters for calculating work share deviation with moving average

ning query to enforce the target share. The share of this user is closer now to the share of the round robin scheduler. A large window size leads to less adjustments of priority and therefor takes longer to converge, but is more robust to changes in the workload.

Choosing the parameters depends on the number of concurrent connections and task sizes and is thus workload dependent. To adapt to changing workloads the scheduler has to observe these parameters and adjust accordingly. The goal for the scheduler is then to chose the interval to be large enough to include a significant number of tasks from each active session, allowing to determine a trend of the work share applying the current set of priorities. It is important to mention, that it does not dependent on the entire runtime of the issued queries. The window size has to be chosen based on the average number of tasks a session executes per interval and the average execution time per task. For shorter sessions, e.g. occurring in interactive applications, a smaller window size helps to quickly balance the load and avoid that one session gets too many resources.

6 Related Work

Workload management for heterogeneous queries has been frequently in the context of web requests [BSUK07, MSAHb03, SHBI+06] and business intelligence applications [BCD+92, KDW+10]. In contrast to our research, most work on workload management was specific to disk-based DBMS and considered a query as the level for scheduling. In general, we can divide the proposed approaches for managing workloads of different query classes into two classes: *external* and *internal*. The general idea of external workload management is to control the number of queries that access the database (admission control). Internal workload management systems typically control available resources, such as CPU or main memory, and assign them to queries. Niu et al. [NMP09] give a more detailed overview of workload management systems for DBMS.

Early work on internal workload management has been published by Carey et al. [BCD+92, CJL89]. The simulation studies are specific to disk-based DBMS, as they extensively model disk-based DBMS characteristics such as disk rotation time or buffer management. A more recent work by McWherter et al. [MSAHb03] shows the effectiveness of scheduling bottleneck resources using priority-based algorithms in a disk-based DBMS. Narayanan et al. [NW11] propose a system for dynamic prioritization of queries to meet given priorities for query classes. In contrast to our work these approaches rely on multiplexing threads on a number of processing units and achieve a targeted resource share either centrally, by prioritizing threads on OS-level or collaboratively, by letting each thread check its consumed resources regularly and sleeping if a certain quota has been met. These strategies are unfavorable for in-memory databases on multi-cores, as execution time is largely dominated by cache locality which is disrupted by context switches.

More recent work has proposed solutions for adaptive admission control based on query response time. Schroeder et al. [SHb06, SHBI$^+$06] propose an external queue management system that schedules queries based on defined service-levels per query-class and a number of allowed queried in the database, the so-called multiprogramming level. Niu et al. [NMP09] propose a solution that manages a mixed workload of OLTP and OLAP queries by controlling to OLAP queries depending on the response times of OLTP queries. Krompass et al. [KKW$^+$10] extended this approach for multiple objectives. The work of Kuno et al. [KDW$^+$10] and Gupta et al. [GMWD09] propose mixed workload schedulers with admission control based on query run-time prediction. Although external workload management systems are applicable to in-memory databases, they fall short in our scenario, as queries need to get access to a large number of processing units quickly, e.g. to answer complex interactive queries.

Until recently, scheduling in operating systems and query scheduling in database management systems were working very differently since queries in DBMS cannot be as easily preempted and were typically very monolithic. With modern many-core systems, task-based decomposition gives the scheduler in DBMS more flexibility and we are able to adapt concepts like [XWY$^+$12] to allow fair scheduling of tasks in IMDBMS.

7 Conclusion and Future Work

In this paper, we have shown that a dynamic priority-based query scheduling can be effectively applied for IMDBMS to fairly schedule mixed enterprise workloads. We are planning to further evaluate the performance of our scheduling approach and extend TAMEX to leverage further information about task characteristics in scheduling decisions. We are further planning to take resource requirements besides CPU, such as cache and memory bandwidth into account to place tasks in a way that will minimize resource conflicts.

References

[BCD$^+$92] K Brown, M Carey, D DeWitt, M Mehta, and F Naughton. Resource allocation and scheduling for mixed database workloads. *cs.wisc.edu*, Jan 1992.

[Bro04] R.G. Brown. *Smoothing, Forecasting and Prediction of Discrete Time Series*. Dover Phoenix editions. Dover Publications, 2004.

[BSUK07] Ernst W. Biersack, Bianca Schroeder, and Guillaume Urvoy-Keller. Scheduling in practice. *SIGMETRICS Performance Evaluation Review*, 34(4):21–28, 2007.

[BTA13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer szu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. ICDE '13. IEEE Computer Society, 2013.

[CJL89] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. VLDB, pages 397–410, 1989.

[GKP$^+$10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: a main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, November 2010.

[GMWD09] Chetan Gupta, Abhay Mehta, Song Wang, and Umesh Dayal. Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. EDBT '09, pages 696–707, New York, NY, USA, 2009. ACM.

[KDW$^+$10] Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Kevin Wilkinson, Archana Ganapathi, and Stefan Krompass. Managing Dynamic Mixed Workloads for Operational Business Intelligence. In *DNIS*, pages 11–26, 2010.

[KKG$^+$11] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Pradeep Dubey, Hasso Plattner, and Alexander Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB, Volume 5, No. 1*, 2011.

[KKW$^+$10] Stefan Krompass, Harumi Kuno, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. Adaptive query scheduling for mixed database workloads with multiple objectives. DBTest '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM.

[LKA04] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[MSAHb03] David T Mcwherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-balter. Priority Mechanisms for OLTP and Transactional Web Applications. pages 535–546, 2003.

[NMP09] Baoning Niu, Patrick Martin, and Wendy Powley. Towards Autonomic Workload Management in DBMSs. *Journal of Database Management*, 20(3):1–17, 2009.

[NW11] Sivaramakrishnan Narayanan and Florian Waas. Dynamic prioritization of database queries. ICDE '11, Washington, DC, USA, 2011. IEEE Computer Society.

[Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. SIGMOD, pages 1–2, 2009.

[Pla11] Hasso Plattner. SanssouciDB: An In-Memory Database for Processing Enterprise Workloads. In *BTW*, pages 2–21, 2011.

[SHb06] Bianca Schroeder and Mor Harchol-balter. Achieving class-based QoS for transactional workloads. *In Proc. of IEEE ICDE*, pages 153–, 2006.

[SHBI$^+$06] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. *Data Engineering, International Conference on*, 0:60, 2006.

[WGP13] Johannes Wust, Martin Grund, and Hasso Plattner. TAMEX: a Task-Based Query Execution Framework for Mixed Enterprise Workloads on In-Memory Databases. In *Workshop on In-Memory Data Management, INFORMATIK, Koblenz (accepted for publication)*, 2013.

[XWY$^+$12] Di Xu, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Zhenjiang Wang. Providing fairness on shared-memory multiprocessors via process scheduling. In *ACM SIGMETRICS/PERFORMANCE*, SIGMETRICS '12, pages 295–306, New York, NY, USA, 2012. ACM.