TAMEX: a Task-Based Query Execution Framework for Mixed Enterprise Workloads on 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 (DBMS) have been proposed to run transactional and analytical applications on a single database instance and to reduce the execution time of complex analytical queries to seconds. The two main reasons for this dramatic performance increase are massive intra-query parallelism on many-core CPUs and primary data storage in main memory. The benefits of these in-memory DBMS for enterprises are huge: analytical applications become largely independent of data staging delays, opening the way for real-time analytics. However, this promising approach will only be adopted, if DBMS can execute dynamically arriving transactional queries in a timely manner, even while complex analytical queries are executed. We believe that two system properties are key to achieve this objective: (1) splitting queries into fine granular atomic tasks and (2) efficiently assigning these tasks to a large number of processing units, thereby considering priorities of query classes. In this paper, we propose TAMEX, a framework for the execution of multiple query classes, designed for executing queries of heterogeneous workloads of enterprise applications on in-memory databases. The basic idea is to generate a task graph for each query during query compilation and assign these tasks to processing units by a user-level scheduler based on priorities. We evaluate the concept using a mix of transactional and join-heavy queries and focus on the impact of task sizes on load balancing and responsiveness of the system.

1 Introduction

In-memory databases 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. Additionally, the dramatic performance increase for complex analytical queries opens the way for a new class of analytical applications, often referred to as *Operational Business Intelligence*. This class of applications leverages the possibility of executing even complex queries with response times in the range of seconds, thus allowing integration of analytics into interactive and mobile applications. Examples for such applications are providing customer specific product recommendations based on historic sales events [WKB+11] or

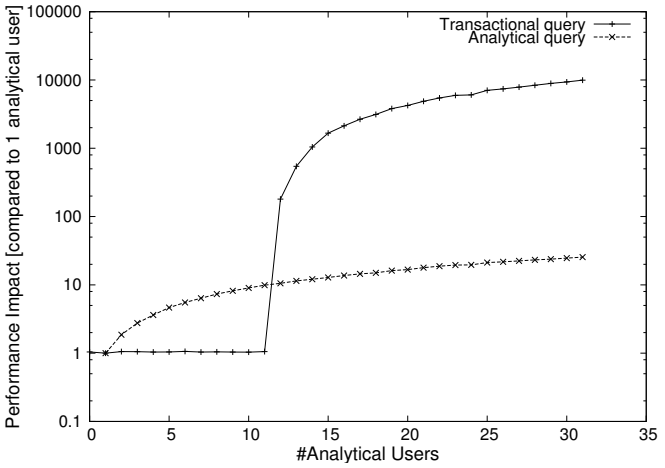


Figure 1: Performance impact of the number of analytical users in MonetDB

real-time checks for product availability [TMZP11]. Running these types of applications on the same database instance that records business events leads to a mix of heterogeneous queries. Depending on the applications issuing the queries, queries may have different response time objectives.

Although queries are executed very fast on IMDBMS, execution time is still constrained by bottleneck resources such as CPU cycles or main memory access. This can potentially lead to resource contention in the DBMS. We distinguish two main causes for resource contention: (1) total load exceeds the capacity of the database, and (2) the database is busy executing low priority queries, while higher priority queries have to wait for execution. The first cause can be eliminated by either adjusting capacity or restricting the number of queries that enter the database, so called admission control. In this paper we optimize for the second scenario by proposing a system for query execution that manages workloads with different priorities, thereby considering specific characteristics of IMDBMS.

As a motivating experiment, we have tested such a scenario with MonetDB [BKM08], a DBMS targeted for analytical workloads. The test machine was equipped with 2 Intel(R) 5670 CPUs with 6 cores each and 144GB RAM. In our experiment, we had one user constantly issuing a transactional query without think time over 120 seconds. Additionally, we had a varying number of n users issuing an analytical query, consisting of a join statement of a large table with 30 million records[1]. Figure 1 shows the performance impact of running additional analytical users on the average response time of queries over the test run as a factor of the average response time for one user of each class. The average response time of transactional queries gets dominated by a queueing delay, when the number of parallel executed analytical queries increases, leading to a drop of transactional query throughput. The dramatic increase at 12 concurrent analytical users correlates with the number of cores available on the test machine. Our objective is to provide a workload

[1]The transactional query equals Q_1 and the analytical equals Q_3 of the evaluation in Section 6

management system that minimizes the impact of analytical queries on the response time of transactional queries.

Workload management for heterogeneous queries is a frequently discussed problem many database administrators struggle with. We believe that the currently proposed solutions fall short for managing a mixed workload on highly parallel IMDBMS (see Section 7 on related work), as they cannot adapt fast enough to provide a larger number of processing units to parallelized queries quickly. As researchers in the parallel computing community [ABD+09], we believe that query execution should be designed for an arbitrary number of processing units and support independent task parallelism to take advantage of massively parallel processors. Therefore, we propose TAMEX in this paper, a TAsk-based framework for Multiple query class EXecution on IMDBMS. The overall idea is to translate arriving queries into a directed acyclic graph of tasks and schedule these tasks based on priorities. We have implemented this approach based on the in-memory storage engine HYRISE [GKP+10].

The remainder of the paper is structured as follows: In the next section, we discuss characteristics of queries in enterprise applications and formulate design objectives for our workload management system TAMEX. Section 3 introduces the assumed underlying system model of the IMDBMS. In Section 4, we motivate a task-based execution of queries. Section 5 introduces our proposed workload management system for multiple query classes TAMEX, which is evaluated in Section 6. The next section discusses related work and the last section closes with some concluding remarks and directions for future work.

2 Challenges of Managing Mixed Database Workloads of Enterprise Applications

In this section, we discuss different query classes that we observe in modern enterprise applications on IMDBMS and derive a list of design objectives that a mixed workload management system should follow. Since productive database systems may differ in terms of number of concurrent users, applications, or service level agreements, these design objectives may generalize specific topics. However, we consider this a necessary step towards understanding the consequences of running multiple applications on a single database.

2.1 Analytical and Transactional Queries

In the domain of enterprise applications, it is commonly distinguished between analytical and transactional applications. Both types of applications issue a specific workload to the underlying DBMS. Although not formally defined, it is widely agreed that transactional queries are considered as short-running and update-intensive, whereas analytical queries are typically being classified as long-running and mostly read-only. Due to their specific fields of application, they also differ in their service level objectives: Transactional applications are often coupled to customer interactions or other time-critical business processes

and have high response time requirements. On the other hand, analytical applications that generate reports are typically considered batch-oriented with comparably less critical response-time requirements.

2.2 Mixed workloads

As motivated in the Introduction, we see a trend towards a mix of analytical and transactional workloads. However, a mixed workload is not a mere combination of analytical and transactional queries.

Using dynamic views in IMDBMS instead of materialized aggregates has been proposed in [Pla09] to allow flexible ad-hoc queries and to simplify database schemas. Due to this design decision, reads of formerly pre-aggregated values are replaced by analytical-style queries that calculate the dynamic views on the fly. To maintain the read performance of materialized aggregates, these on-the-fly aggregations have to be executed immediately, introducing a new query class with high complexity and strict response time characteristics.

As discussed in the Introduction, providing a platform that can run a mixed transactional and analytical workload on a common dataset opens the way for new business applications. Krueger et al. [KTG+10] give a detailed overview of the characteristics of these mixed workload applications. Again, these applications introduce queries with a complexity typical for analytical applications in a transactional context.

In summary, we see queries with a complexity typically found in analytical queries, but stricter response-time requirements. In the following, we will refer to these queries as *transactional analytics*.

2.3 Query Classes

Summarizing the previous sections, we can derive three main query classes that need to be executed in an IMDBMS in a mixed workload scenario:

Transactional Queries low complexity, strict response time requirements in the order of milliseconds, sequential execution possible

Transactional Analytics high complexity, strict response time requirements in the order of seconds, parallel execution mandatory

Batch-oriented Analytics and Maintenance Tasks high complexity, weak response time requirements, parallel and serial execution

We have categorized query classes into complexity, response time requirements and type of execution. The type of execution indicates the preferred execution model: transactional queries typically read or write a limited number of records; these tasks are typically too

fine-granular to be split into several subtasks for parallel execution, as the overhead for splitting the tasks is to high. On the other hand, transactional analytics typically process a large number of records. Partitioning data and parallel execution of sub tasks is mandatory to meet strict response time requirements. Orthogonal to this classification, queries can have priorities depending on the importance of the application or user that issued a query.

2.4 Design Objectives for Mixed Database Workload Management

Based on the query classes defined in the previous section, we can state a number of desirable design objectives for an IMDBMS executing a mix of these query classes. First, queries need to run in parallel to efficiently use available resources: as transactional queries typically run sequentially, allowing only one query at a time would result in an underutilization of the system. Also, analytical queries or other database tasks will have parts that cannot be parallelized to a degree that leverages all available processing units. Second, transactional queries need to run with a higher priority over other query classes: given the relatively short run-time of transactional queries compared to the other classes, they cannot wait until queries of the other classes have completed to meet their service level objectives. Third, transactional analytics have to gain access to a large number of resources quickly: given the tight response time requirements and the high complexity, queries of type transactional analytics need to be executed with a high degree of intra-query parallelism and therefore require access to a larger number of available processing units. And fourth, if a query is running on a number of processing units in parallel, it needs to free up resources quickly in case a query with higher priority arrives for execution.

3 System model

This section briefly describes our system architecture and transaction model. Although our proposed approach for query execution is largely agnostic to specific architectural details of the database, it is designed for an IMDBMS with characteristics in mind as described in the following . As an example, we assume a decomposed or column-oriented storage model, which is well suited for intra-operator parallelism [Pla09]. Changing the storage layout may affect the ability of partitioning the input data of an operator and executing the operator with multiple threads.

We assume an in-memory database following the system model described in [Pla11], where data is physically stored decomposed in a column-oriented structure. All columns are dictionary-compressed to utilize memory and bandwidth efficiently. While column-orientation typically favors read-mostly analytical workloads, updating and inserting data into dictionary-compressed column structures is challenging. 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 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. TIDs issued by a transaction manager for each arriving query define the start order of transactions. See [Pla11] for more details. To achieve durability in case of a system failure, the IMDBMS writes log information to persistent storage. In [WBR+12], we describe an efficient way of logging dictionary compressed columns.

4 Task-Based Query Execution

Task-based execution is defined as a transformation of the logical query plan into a set of atomic tasks that represent this plan. These tasks may have data dependencies, but can be executed otherwise independently, thus forming a directed acyclic graph. The general execution model is comparable to MonetDBs[BKM08] bulk-execution model, with the important difference that intermediate results are not necessarily materialized but can be either materialized rows or record positions in the table. In this section, we describe the rationale of applying a task-based approach, the mapping of a query to a task graph, and how parallel execution of tasks is achieved.

4.1 Motivation for Task-Based Execution

A task-based approach provides two main advantages over considering a whole query as the unit for scheduling: More fine granular units of scheduling allow for better load balancing on a multiprocessor system, as well as more control over progress of query execution based on priorities.

We assume that our IMDBMS runs on a hardware platform with high number of processing units. Today's commodity servers are equipped with up to eight CPUs, each CPU being "multicore" with up to 10 cores and the number of cores per CPU is expected to grow even further [ABD+09]. Following this observation, we apply a task-based approach to query execution as the foundation to scalable query execution on multicore platforms. Compared to assigning a fixed number of processing resources to the execution of a single query, this strategy enables dynamic load balancing [BFV96], which also works effectively if execution time of tasks cannot be predicted precisely.

In addition to improved load balancing, splitting queries into small units of work introduces points in time 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.

4.2 Mapping Queries to Task Graphs

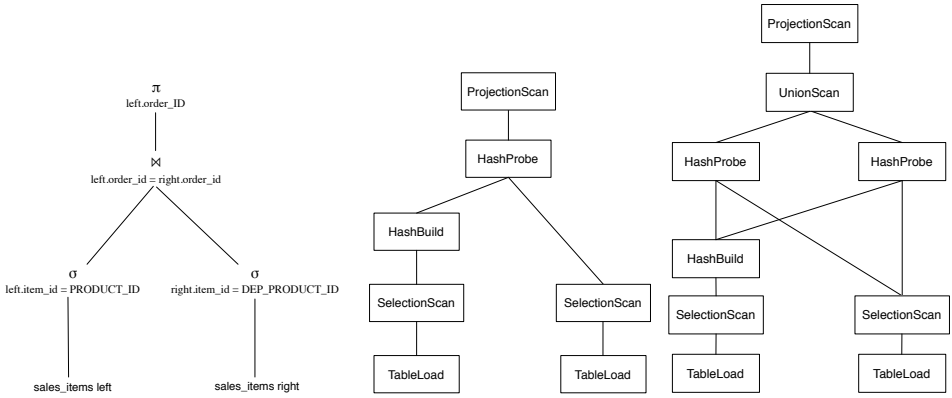
A query executed by a relational database is typically expressed as a physical query execution plan forming a tree of relational operators. We consider an optimal query plan as given; compile-time query optimization is orthogonal to this work. A task graph is directly derived from an execution plan similar as proposed by Krikellas et al. [KCV10]. This process involves an expansion of relational operators into a number of sub-operators depending on the chosen algorithm. E.g., a hash join is expanded in a hash build and a probe operation.

The data dependencies expressed as edges in the query plan are taken over in the task graphs: an edge in the task graph means that a task can only start execution if all preceding tasks have been executed —such a task is called ready.

As an example, consider the self join of an order item table described in [WKB+11]. The query determines the orders that contain two given products `PRODUCT_ID` and `DEP_PRODUCT_ID`, which are required to calculate the probability that two products have been bought together.

```
SELECT letf.order_ID                                    1
FROM                                                    2
    sales_items left,                                   3
        sales_items right                               4
      WHERE                                             5
            left.order_ID = right.order_ID              6
      AND                                               7
            left.item_ID = PRODUCT_ID                   8
      AND                                               9
            right.item_ID = DEP_PRODUCT_ID              10
```

We assume a database schema, where orders are stored in a table with all the relevant order information and the ordered items are stored in a separate item table that holds an `order_ID` as foreign key to the header table. The corresponding relational query plan is illustrated in Figure 2(a). Figure 2(b) shows the corresponding task graph using the operator names given in our implementation. Given this task graph, execution would start with the two `TableLoad` operators, that get a reference to the corresponding tables and load them to main memory, if not loaded yet. Once table `sales_items` is loaded, the operator `SelectionScan` starts. We consider a task an atomic unit of scheduling, hence, a task is processed by a single processing unit. It is important to mention that the projection is not pushed further down in the task graph before the join execution as we do not materialize intermediate results in this plan. The projection is only required for implicit result materialization, but does not materialize the data itself.



(a) Query execution plan (b) Task graph (c) Parallel task graph

Figure 2: Mapping of task query execution plan to task graph

4.3 Parallel Execution of Task Graphs

As discussed in Section 2, parallel execution of queries is necessary to achieve the desired performance characteristics. In query execution, we typically differentiate three forms of parallelism: (1) inter-query parallelism, (2) inter-operator parallelism and (3) intra-operator parallelism. Our execution framework supports all three.

Inter-query parallelism is achieved by executing several task graphs concurrently. Independent tasks can run concurrently on different processing units. Intra-query parallelism is achieved by running all ready tasks of a single query in parallel. Intra-operator parallelism is achieved by splitting a task into several tasks, while partitioning the input data. This is achieved by rewriting the task graph and replacing tasks by a number of subtasks. As example, Figure 2(c) shows a version of the task graph of Figure 2(b) with the *HashJoin* operator split into two tasks. Note that additional to the sub tasks that perform the hash join, an additional operator that combines the results—in this case a union—is added to the graph.

5 Managing Mixed Enterprise Workloads with TAMEX

This section gives an overview of the task-based query execution framework TAMEX, which is implemented based on HYRISE [GKP+10]. We extended HYRISE to support parallel execution of queries, as well as intra-query parallelism, based on multithreading. Each operator can be split into subtasks to support intra-operator parallelism as described in the previous section. Additionally, we implemented a priority-based task scheduler.

Figure 3 gives a conceptual overview of TAMEX. Notice that an external admission control that defers or rejects queries that cannot be handled by the available resources efficiently is

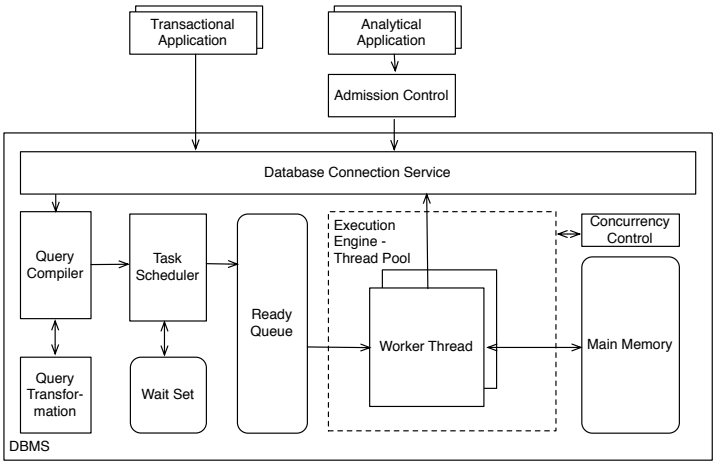


Figure 3: Architectural overview of TAMEX

complimentary to our work (see Section 7 on related work). As discussed in the introduction, this work focuses on resource contention caused by executing low priority queries, while higher priority queries have to wait for execution.

Inside the database, we have a service for connection handling, which forms as the interface to user applications. In the current implementation, a user request consists of a hand-coded query execution plan, defining operators and data dependencies. The plan is annotated with query priority and degree of intra-operator parallelism per operator. The *Query Compiler* takes incoming requests and creates a task graph for each incoming query. The *Query Transformation Engine* rewrites the task graph based on the provided annotations. As an example, the *Query Transformation Engine* rewrites the task graph illustrated in Figure 2(b) to the graph illustrated in Figure 2(c), if a degree of parallelism of two is provided for operator *HashProbe*. It is important to mention that even though an operator can be annotated to split operators into several tasks, the execution of a task is single-threaded so that the scheduler has full control over all resources.

Task execution is implemented using a threadpool. The *Task Scheduler* takes task graphs and places tasks into the *Ready Queue* if tasks have no unmet dependencies, otherwise in the *Wait Set*. *Worker Threads* run an infinite loop of taking and executing tasks from the *Ready Queue*. If a task has been executed, it notifies dependent tasks; once all dependencies are met, a task is moved from the *Wait Set* to the *Ready Queue*. The task scheduling framework is implemented in a modular way so that the *Task Scheduler* can be replaced easily to implement different scheduling strategies.

Here, we present a priority-based task scheduler to handle a mix of query classes as defined in Section 2.3. This scheduler implements the *Ready Queue* of Figure 3 as a priority queue with equal worker threads that execute tasks of any priority. We have used a lock based priority queue to synchronize concurrent access; we have also implemented a distributed priority queue with work stealing, but only show the results of the global queue

here, as it did not become a bottleneck and outperformed the distributed queue. Dragicevic et al. [DB08] give an overview of alternative implementations. The order criteria of the priority queue is a tuple of priority and transaction id (TID) (as introduced in Section 3), where tasks are first ordered based on priority and, secondly, based on TID. TID as secondary sort key is introduced to support a FIFO scheduling behavior for queries with similar priority. This design strictly prioritizes queries with higher priority over queries with lower priority, potentially leading to starvation of lower priority tasks. This approach can be easily extended to have a queue for each class (quantized queues [DB08]) with task selection based on a predefined share for each query class to ensure progress for all query classes. But even despite strict priorities, lower priority queries can interfere with higher priorities as tasks of lower priorities need to finish before a higher priority task can be scheduled —the impact depends on the task sizes as shown in the next section. As transactional queries are defined to run sequentially, we implemented them as a single task to avoid this potential impact of lower priority tasks. To minimize the effect of lower priority on higher priority queries, task sizes of lower priority queries should be chosen small—clearly, there is a trade-off between a small size to support effective load balancing and additional overhead of task creation.

6 Evaluation

This section provides an experimental validation of the effectiveness of our workload management system TAMEX. The test machine was equipped with 4 Intel(R) Xeon(R) X7560 CPUs with 8 cores each and 512GB RAM. We ran TAMEX with 32 threads, each thread pinned to a core; whether additional threads can further improve performance using simultaneous multithreading (SMT) depends on operator implementations, as discussed in [CB13] for joins. We have defined a characteristic query for each of the three query classes defined in 2.3. The transactional query Q_1 is a select statement to get the address information for a customer id from an address table. For the class transactional analytics, we chose the query described in Section 4, but replaced the hash join implementation with a better scalable radix join implementation [CB13]; we call this query Q_2. As a lower priority analytical query (Q_3), we chose a query similar to query 6 of the ch-benchmark [CFG+11], adapted to the sales and header table layout described in Section 4, which involves an additional join of headers and items [2]. The sales header table has 13.7 million records, the sales items table 30 millions records and the customer table 18,500 records. In our setup, a user constantly issues one of the three queries for 120 seconds for each test run. The user that simulates transactional analytics (Q_2) has a think time of one second, before issuing a new query. For Q_2 and Q_3, we chose a degree of parallelism of 32, so that each query can fully utilize all cores of the machine. With this degree of parallelism, Q_2 leads to a task graph of 394 tasks and Q_3 of 426 tasks.

Figure 4 shows our results with TAMEX with and without the use of priorities for a single user issuing query Q_1 and Q_2 and an increasing number of Q_3 users. The graph shows the

[2] SELECT SUM(h.amount) FROM sales_header h, sales_item i WHERE h.order_date >= "1999-01-01" AND h.order_date < "2020-01-01" AND i.quantity between 1 and 100000 AND h.order_ID = i.order_ID

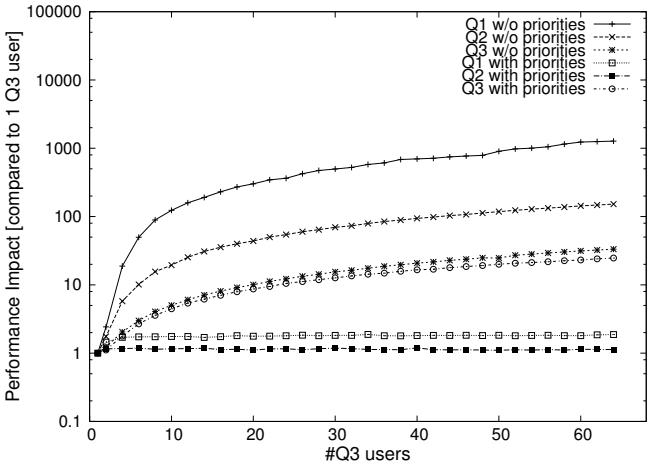


Figure 4: Performance impact of varying the number Q_3 users with TAMEX

performance impact of the number of Q_3 users by dividing the average response time of a test run for each query class by the average response of that class with one Q_3 user. Without priorities, the response times for Q_1 and Q_2 increase with the number of additional Q_3 users. This is comparable to the behavior of MonetDB; the steeper increase with TAMEX below 12 Q_3 users compared to MonetDB is caused by intra-operator parallelism for analytical queries with TAMEX. A smaller number of concurrent analytical queries than cores can fully utilize the available processing units. In our priority-based implementation we gave Q_1 the highest priority, followed by Q_2 and Q_3. We see an initial performance penalty for transactional queries. As Q_3 queries occupy are executed on all available resources, incoming higher priority queries may have to wait until one of the tasks of query Q_3 has finished. However, a further increase of the number of analytical users does not further impact the response times for Q_1 and Q_2, which is the desired system behavior we wanted to achieve.

Lower priority queries can delay the response time of incoming higher priority queries, as actual running tasks need to finish to free resources. The impact of this effect depends on the task size. To demonstrate this, we ran an experiment with one user each for queries Q_1 and Q_2 as discussed above and 32 users issuing Q_3 queries with a varying degree of intra-operator parallelism. For a degree of 64, each Q_3 query generates a task graph consisting of 842 tasks. A higher degree of intra-operator parallelism leads to smaller task sizes. We chose 32 analytical users to fully utilize the test machine, even in the case of no intra-operator parallelism for Q_3. Figure 5 shows the change of the average response times for queries Q_1 and Q_2 as a percentage of a degree of 32. With no intra-operator parallelism, we see the highest average response time for transactional queries (60% higher) and transactional analytics (almost 140% higher), as potentially all cores are occupied with long-running tasks, if a higher priority query arrives. This effect weakens with a decreasing task size, leading to shorter average response times, if lower priority queries have smaller task sizes.

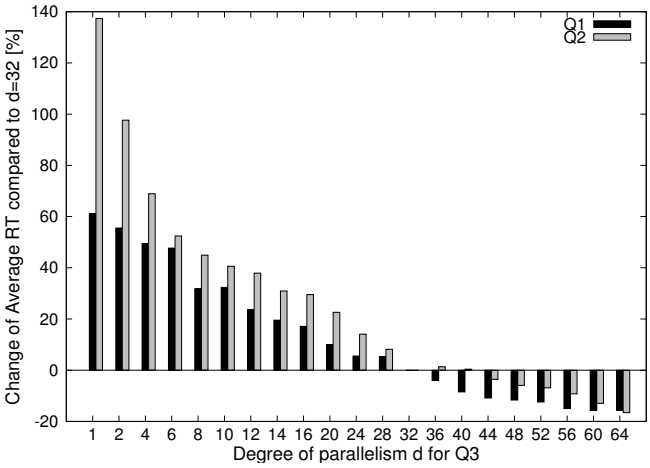


Figure 5: Performance impact of varying task sizes of Q_3

7 Related Work

We have identified two main areas of work related to our research: workload management for DBMS and query scheduling for multiprocessors. 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.

7.1 Workload Management in DBMS

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.

More recent work 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. [NMP+07] 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 as well, 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 quickly.

7.2 DBMS and Query Execution on Multiprocessors

Our framework is designed for parallel query execution on multiprocessor CPUs. Hardavellas et al. have studied the opportunities and limitations of executing of database servers on multiprocessors [HPJ+07], concluding that data cache misses are the performance bottleneck resource in memory-resident databases. Zhou et al. [ZCRS05] investigated the impact of simultaneous multithreading.

Furthermore, a related field is query execution on multiprocessors. Krikellas et al. [KCV10] and Wu et al. [WCwHK04] discuss the question of how a single query plan can be scheduled on a number of given threads. Wu et al. [WCwHK04] also propose an extension to dynamic arrival of queries. Both approaches apply heuristics to generate a schedule with minimal makespan for a given query, but the algorithms seem to impose significant overhead in case of a larger number of tasks and threads.

Bouganim et al. [BFV96] as well as Rahm et al. [RM95] have studied load balancing strategies for parallel database systems. In the area of task based scheduling for queries, Lu et al. [LT92] propose a task-based approach for query execution with dynamic load balancing by task stealing. In contrast to our approach, they schedule tasks of larger size that can be further split during runtime in case of load imbalances.

8 Conclusion and Future Work

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

References

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[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.

[BFV96] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. Dynamic Load Balancing in Hierarchical Parallel Database Systems. VLDB '96, pages 436–447, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[CB13] Gustavo Alonso Cagri Balkesen, Jens Teubner and M. Tamer szu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. *In Proc. of IEEE ICDE*, 2013.

[CFG+11] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload CH-benCHmark. DBTest '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM.

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

[DB08] Kristijan Dragicevic and Daniel Bauer. A survey of concurrent priority queue algorithms. pages 1–6. IEEE, 2008.

[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.

[HPJ+07] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 79–87, 2007.

[KCV10] Konstantinos Krikellas, Marcelo Cintra, and Stratis Viglas. Scheduling threads for intra-query parallelism on multicore processors. In *EDBT*, 2010.

[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.

[KTG+10] Jens Krueger, Christian Tinnefeld, Martin Grund, Alexander Zeier, and Hasso Plattner. A Case for Online Mixed Workload Processing. In *DBTest*, 2010.

[LT92] Hongjun Lu and Kian-Lee Tan. Dynamic and Load-balanced Task-Oriented Datbase Query Processing in Parallel Systems. In *EDBT*, pages 357–372, 1992.

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

[NMP+07] Baoning Niu, Patrick Martin, Wendy Powley, Paul Bird, and Randy Horman. Poster Session: Adapting Mixed Workloads to Meet SLOs in Autonomic DBMSs. 2007.

[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.

[RM95] Erhard Rahm and Robert Marek. Dynamic Multi-Resource Load Balancing in Parallel Database Systems. VLDB '95. Morgan Kaufmann Publishers Inc., 1995.

[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.

[TMZP11] Christian Tinnefeld, Stephan Mueller, Alexander Zeier, and Hasso Plattner. Available-To-Promise on an In-Memory Column Store. In *BTW 2011*, 2011.

[WBR+12] Johannes Wust, Joos-Hendrik Boese, Frank Renkes, Sebastian Blessing, Jens Krueger, and Hasso Plattner. Efficient Logging for Enterprise Workloads on Column-Oriented In-Memory Databases. In *CIKM*, 2012.

[WCwHK04] Jun Wu, Jian-Jia Chen, Chih wen Hsueh, and Tei-Wei Kuo. Scheduling of Query Execution Plans in Symmetric Multiprocessor Database Systems. In *IPDPS*, 2004.

[WKB+11] Johannes Wust, Jens Krger, Sebastian Blessing, Cafer Tosun, Alexander Zeier, and Hasso Plattner. xSellerate: Supporting Sales Representatives with Real-Time Information in Customer Dialogs. In *In-Memory Data Management*, 2011.

[ZCRS05] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. Improving database performance on simultaneous multithreading processors. VLDB '05, pages 49–60. VLDB Endowment, 2005.