QTM: Modelling Query Execution with Tasks

Steffen Zeuch Johann-Christoph Freytag

Department of Computer Science, Humboldt-Universität zu Berlin
{zeuchste,freytag}@informatik.hu-berlin.de

ABSTRACT

Over the last decade, several approaches for parallel query execution have emerged. The performance of these approaches is mainly affected by the non-manageable cache hierarchy. However, each approach exploits the capabilities of modern processors differently. Furthermore, the comparison is difficult due to different operator-to-resource assignments during run-time (scheduling strategy) and the number of tuples each operator processes (chunk size).

In this paper, we first classify common DBMS by their scheduling strategies and chunk sizes. Then, we propose a task model called *Query Task Model (QTM)* that opens a design space for database schedules. With QTM, we generalize the modeling of parallel query execution such that different approaches become comparable. Using QTM, we model an arbitrary QEP as a set of tasks. Each task represents a particular piece of work on a subset of data.

Our evaluation of different schedules modeled in QTM shows, that a tuple-at-a-time schedule cannot exploit modern hardware efficiently. In contrast, an operator-at-time schedule increases the performance due to increased cache utilization. However, a buffer-at-a-time schedule that takes the cache hierarchy into account outperforms schedules that do not. Furthermore, we show that a schedule that is optimized for data cache locality does not necessarily outperform a schedule optimized for instruction cache locality. We identify a sweet spot where the ratio of data locality and instruction locality produces the fastest schedules.

1. INTRODUCTION

Over the last decades, the clock speed per core reached a plateau due to several physical limitations. Since then, an increasing number of available on-chip transistors are used to incorporate more processors and larger cache. Additionally, a large amount of commonly available main memory allows modern database management systems (DBMS) to store their entire working sets in main memory. Nowadays, CPUs process data much faster than transferring data from main memory into caches. This trend creates a *Memory Wall* which is the main challenge for modern main memory database systems [1, 4].

Proceedings of the VLDB Endowment, Vol. 7, No. 14
Copyright 2014 VLDB Endowment 2150-8097/14/10.

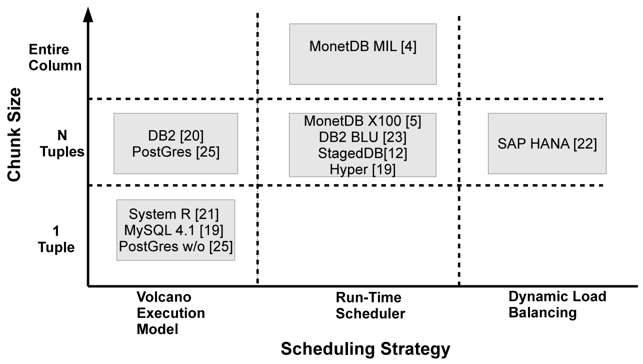


Figure 1: Classification of Databases.

Research in the last decade also shows, that parallelization and chip multiprocessing exacerbate these *Memory Wall* [1, 4]. The ever increasing number of processing units per chip have to share a constant memory bandwidth; thus, reducing the available memory bandwidth per processing unit. An uncoordinated parallel access to shared data structures from different processing units leads to a *memory bottleneck* [4]. To overcome the memory bottleneck, the locality of data and instructions become increasingly important. The cache hierarchy of modern processors alleviates the memory bottleneck by reusing already loaded data and instructions in caches. However, caches cannot be controlled directly. Thus, a database might only guide cache behavior by indirect means like data placement and access patterns. The exploitation of these indirect means are vital for chip multiprocessors to supply each CPU with sufficient data despite limited main memory bandwidth.

Due to their inherent parallelism, the opportunities of chip multiprocessors are in particular applicable in DBMS. On the other hand, a DBMS may also exhibit tight data dependencies that require some degree of synchronization between operators executed in parallel. To exploit parallelism in databases, different approaches for parallel query execution have emerged over the last decade. In Figure 1, we classify common databases by their scheduling strategy and chunk size. The scheduling strategy controls the processing of different operators by different processors. The chunk size determines the number of tuples processed by each operator instance and ranges from one tuple, over multiple tuples (N), to an entire column.

In the light of this discussion, our contributions are as follows:

- We classify common DBMS by their scheduling strategies and chunk sizes.
- We define QTM (Query Task Model), a model that allows to express and compare different approaches for parallel query execution.
- Using QTM, we compare common query execution schedules regarding their cache utilization.
- Based on our analysis, we identify a sweet spot that produces the fastest schedules.

The remainder of this paper is organized as follows. In Section 2, we classify common databases by their scheduling strategies and chunk sizes. In Section 3, we introduce our *Query Task Model (QTM)*. In Section 4, we model common database schedules with QTM. In Section 5, we present our evaluation results and show related work in Section 6. We conclude and outline future work in Section 7.

2. CLASSIFICATION OF DATABASE SCHEDULERS

Over the last decades, different approaches for parallel query execution have emerged because scheduling a database query exhibits some degrees of freedom. In the following, we present four alternatives a database scheduler might exploit when optimizing query execution with respect to available resources.

First, a scheduler has to determine a execution order for available operators among queries. For a single query, the execution order has to satisfy the constraints introduced by a QEP. For multiple queries, the execution order for pending queries has to take fairness and priorities into account. Second, a scheduler has to assign a degree of parallelism (dop) to each operator. A dop can be either determined statically during compile-time or dynamically during run-time. Third, a scheduler has to specify a degree of thread cooperation. In general, threads can either work cooperatively on the same operator or separately on different operators. Finally, a scheduler has to partition the input for each operator. The size of a partition, so-called *chunk size*, determines how many tuples are processed by an operator before returning the result.

In this paper, we focus on operator scheduling for a single query during run-time. Furthermore, we assume an invariable QEP that was generated by a query optimizer as an input. In the following, we classify different database scheduler by their scheduling strategies and chunk sizes as shown in Figure 1. At first, we show how early databases implement scheduling before introducing three classes of state-of-the-art database schedulers in Section 2.1. After that, we describe chunk sizes that are proposed in research and their impact on query execution in Section 2.2. Finally, we show possibilities for exploiting parallelism in databases as well as different degrees of thread cooperation in Section 2.3.

2.1 Scheduling Strategies

The first dimension for our DBMS classification is based on the applied *scheduling strategy* (see Figure 1). Early DBMS implemented a *static optimization approach* that determined the dop of an operator statically based resource availability at compile-time [21, 27]. During run-time, the dop was implemented by a static assignment of threads to operators. The main disadvantage of this approach is the temporal gap between compile-time and run-time. During run-time, the system load might be quite different which may lead to a suboptimal resource utilization. Furthermore, uncertain information at compile-time such as wrong cardinality estimates, skewed data, correlated attributes, outdated statistics, or user-defined functions, may also lead to a suboptimal decision [14]. The result of these uncertainties could be a wrong prediction of operator work (execution skew) that leads to an imbalanced work distribution. Additionally, a static assignment of threads to operators introduces a *discretization error*. Since operators and processors are discrete entities, a fixed number of operators cannot be assigned to processors such that each operator reaches its optimal dop [6]. Finally, a static assignment may lead to a *pipeline delay problem*. Therefore, processors that are assigned to operators at the end of an execution plan idle at the beginning and processors at the beginning idle at the end [17]. To response to these compile-time uncertainties, three different classes of database schedulers have emerged.

A first class of state-of-the-art database schedulers responds to compile-time uncertainties at run-time. For example, XPRS implements a *two-phase optimization approach* [21, 13]. In the first phase, the optimizer ignores aspects of parallelism and produces the best sequential plan during compile-time. In the second phase, the plan is optimized for parallelism during run-time. Thus, a plan is decomposed into a set of plan fragments and the dop is determined based on the current resource availability. After that, a *parallel executor* determines a processing schedule and distributes the fragments for execution. These approach leads to an improved resource utilization because it takes resource availability at run-time into account to determine the dop. Following this approach, *dynamic optimization* during run-time becomes a vital source for query execution performance and databases implement a dedicated *run-time scheduler*. A run-time scheduler is able to react to changes in the system load or incorrect estimations during run-time. Over the last decades, even more complex scheduler have been proposed [5, 23, 15, 12]. For instance, one run-time scheduler takes NUMA-characteristics into account [15] and another implements a time-slice based scheduling algorithm [12].

A second class of state-of-the-art database schedulers responds to compile-time uncertainties with *dynamic load balancing*. At compile-time, a query plan is disassembled into tasks that are placed into a queue. During run-time, each processor dequeues tasks until the queue is entirely processed. A dynamic load balancing approach omits a dedicated run-time scheduler because the actual mapping of threads to operators is implemented using a *work-pull* strategy. Thus, each processor dynamically acquires new work on its own if computing capacities are available. The execution order of operators is determined by the order of tasks in the queue. The dop of an operator is not statically defined and depends on the number tasks currently executing this operator. Furthermore, predicting which processor executes which task reveals high uncertainties because differently sized tasks and varying resource availabilities introduce high variability. The proposed approaches in research vary in the number of queues and the granularity of tasks [6, 17, 16, 22].

A third class of state-of-the-art database schedulers responds to compile-time uncertainties with a simple execution model. The demand-driven *volcano execution model* emerged as the most commonly used scheduling strategy [10, 25, 19]. This model hides aspects of parallelism from operators and omits a dedicated run-time scheduler. Instead, the volcano execution model implements a *open-next-close* iterator interface for each operator. The *open* call initialize the operator and the *close* call deallocates all resources. The *next* call on one operator propagates recursively to its child operators until one output tuple is generated. Through repeated *next* calls, all tuples are processed and the operator can be closed. The actual assignment of resources to operators is implicitly implemented by the model. Thus, this model makes parallel query execution entirely self-scheduling [10]. The advantages are the avoidance of synchronization and scheduling, minimized data copies, reuse of current data items in main memory, and lazy operator evaluation [25]. Graefe extends this model for parallelism by introducing *exchange* operators to synchronize different threads executing the same query plan [10].

2.2 Chunk Size

The second dimension for our DBMS classification is based on the *chunk size* (see Figure 1). The input of an operator can be divided into multiple chunks (partitions) which can be processed in parallel by different operator instances (intra-operator parallelism). The chunk size determines how many tuples are processed by an operator instance before the result is returned. Thus, the chunk size determines the number of operator calls.

The chunk size is defined either at compile-time, run-time, or as a constant value. However, all DBMS shown in Figure 1 define a fix chunk size for all QEPs. An alternative approach proposed by Cieslewicz et al. [8] suggests to change the chunk size dynamically based on cache miss sampling during run-time.

The chunk size in common DBMS varies significantly. Some approaches define a chunk size in relation to a hardware parameter [28, 20, 7, 25]. However, most approaches state, that they adjust the chunk size such that the entire chunk fits into a certain cache [5, 23]. Thus, common chunk sizes match L1, L2 or L3 cache sizes. Other approaches define a fix number of tuples [15] or a fix block size like 64KB [24]. Two extremes are the chunk size of one tuple used in the classical volcano execution model [10] and the chunk size of one column used in MonetDB/MIL [4].

Block-oriented processing [20] extends the volcano execution model by changing the number of tuples transferred between two operators from one tuple to a *block* or a chunk of tuples. Thus, the data locality is improved and the overhead of one operator call is amortized over multiple tuples. In general, block-oriented processing increases the performance as long as the entire block of tuples fits into a cache [28]. Zhou and Ross [25] implement the block-oriented approach by inserting buffers between certain operators which shows improved instruction cache performance. Zukowski et al. [28] compare the row-wise storage layout (NSM) and column-wise storage layout (DSM) in combination with block-oriented processing. They show, that the storage layout strongly influences the performance of different database operations.

2.3 Degree of Parallelism

A degree of parallelism has to be specified in any scheduling strategy and has to take the type of operators in the QEP into account. Database operators can be classified into *blocking* and *non-blocking* operators. A *blocking* operator, like sort or aggregation, needs to collect all input tuples before it produces the first output tuple. In contrast, a *non-blocking* operator, like selection or hash probe, produces output tuples on-the-fly. A sequence of non-blocking operators in a producer/consumer relationship represents a so-called *pipeline*. Blocking operators are parallelized by exploiting *intra-operator* parallelism. Therefore, the input of a blocking operator is partitioned into chunks and processed in parallel. A scheduler implements intra-operator parallelism by disassembling one operator into multiple operator instances. During run-time, each operator instance processes one chunk. Commonly, the number of instances is determined by $\lceil \frac{number\ of\ input\ tuples}{chunk\ size} \rceil$. In contrast, a pipeline containing only non-blocking operators enables a much higher flexibility for parallel processing. A scheduler can parallelize a pipeline by exploiting intra-operator and *inter-operator* parallelism. Inter-operator parallelism enables parallel processing of different operators without blocking. Finally, *independent parallelism* can be exploited if two pipelines exhibit no dependency. In this case, both pipelines may be processed in parallel [9].

The actual implementation of dop during run-time has to take the degree of thread cooperation into account. In general, threads can be either working cooperatively together on the same operator instance or each threads works on its own operator instance. Cieslewicz et al. [7] propose a parallel buffer that is filled by a group of threads cooperatively before the buffer is delivered to the next operator. Thus, all threads work on the same operator instance. An alternative approach proposed by Cieslewicz et al. [8] implements a strategy where a distinct chunk of tuples is assigned to each thread. Thus, each threads processes one operator instance independently.

3. QUERY TASK MODEL

In this section, we present our *Query Task Model* (QTM) which opens a design space for database schedules. With QTM, we generalize the modeling of parallel query execution such that different approaches become comparable. In Section 3.1, we introduce QTM as a task model for query execution before we describe the implementation of QTM in Section 3.2. After that, we define QTM formally. The formal definition consists of tasks and task configurations (Section 3.3), processing strategies (Section 3.4), and queues (Section 3.5). Finally, we describe different aspects of parallelism in QTM in Section 3.6.

3.1 QTM Overview

The task paradigm is a common abstraction for parallelism in high performance computing [18]. A general task model for parallel computing consists of *tasks*, *units of execution* (UEs), and *processing elements* (PEs). A *task* corresponds a certain part of an algorithm and is implemented by grouping a sequence of instructions. During execution, each task is mapped to a *unit of execution* (UE) that is either a thread or a process. A UE has to be executed by a *processing element* (PE). A PE is a generic term for a hardware unit that is either a processor or a machine [18].

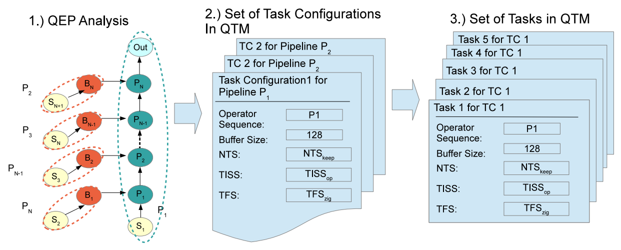


Figure 2: QEP Transformation Process.

We extend this general task model to a model for parallel query execution of database queries (QTM) that allows us to express and compare different approaches. In particular, we have to include database specific constraints and requirements. To execute a database query using a task model, we have to create tasks from a query execution plan (QEP), map tasks to UEs, and schedule UEs by PEs for execution. In QTM, we create tasks by disassembling a QEP at compile-time. Figure 2 illustrates our three-step query transformation process to transform a QEP into a set of tasks that is modeled in QTM. In this section, we describe the transformation process conceptually. We leave the development of a general framework for transforming an arbitrary QEP into QTM tasks as future work. The transformation process in Figure 2 proceeds as follows.

In a first step, we analyse a QEP to identify a set of pipeline fragments with maximal length and generate a dependency graph describing their relationships. The dependency graph reveals ordering constraints between operators. In a second step, we group operators into *task configurations* (TC). Each TC represents a particular piece of work of a QEP on a subset of data. Based on TCs, we describe dependencies and potential concurrent execution for a group of operators. A TC represents a blueprint that specifies the work (operator sequence) and data (buffer size) for its tasks (see Section 3.3). Furthermore, a TC defines three *processing strategies* which specify task execution during run-time (see Section 3.4). The mapping of operators to TCs exhibits some degree of freedom. Possible mappings range from a *fine-grained* mapping of one operator to one TC up to a *coarse-grained* mapping of an entire pipeline to one TC. Note, TCs are only used as an intermediate format during the transformation process.

In a third step, we use TCs to instantiate as many tasks as necessary to process all input tuples. Each task inherits the operator sequence, buffer size, and processing strategies from its task configuration. A task executes its operator sequence for each tuple in its buffer. Additionally, task execution is specified by its inherited processing strategies. Within a task, we encapsulate a particular piece of work of a QEP as an operator sequence and a subset of data as a chunk of tuples in a buffer. The number of tasks per TC is determined by the ratio of input tuples and buffer size. As the result of this transformation process, we obtain a set of tasks that is modeled in QTM. With QTM, we extend the notion of tasks proposed in previous work [6, 17, 16, 22] by a generalized work and data specification and a declaration of processing strategies which specify task execution during run-time.

For the rest of this paper, we assume a QEP as input that is modeled as a set of tasks in QTM. Thus, we focus on comparing different query execution strategies within QTM.

3.2 QTM-DLB

With QTM, we model a QEP as a set of tasks. However, the actual scheduling of these tasks depends on the run-time implementation. A run-time implementation consists of two processing steps. First, it has to establish a particular order between tasks that satisfies the constraints introduced by a QEP. Second, it has to manage task execution following a scheduling strategy.

In this paper, we introduce *QTM-DLB* as a run-time implementation of QTM. QTM-DLB implements a dynamic load balancing (DLB) approach with one global task queue. We decided to implement QTM using a DLB approach because it already based on the notion of tasks. Compared to other DLB approaches [6, 17, 16, 22], QTM-DLB executes generalized tasks specified in QTM. To establish a particular order between tasks, we define a *placement strategy* (PS). In QTM-DLB, we apply a PS as the last step during compile-time to place tasks into the *global task queue*. Note, other possible run-time implementations might use the volcano execution model or a run-time scheduler as the scheduling strategy. Thus, they might specify their run-time query execution with QTM. In this case, one task in QTM might represent a *next call* in the volcano execution model or a *operator call* in a run-time scheduler.

The execution of tasks in a general task model is implemented by a mapping of tasks to UEs. In QTM-DLB, we choose to map tasks to threads because threads of the same process share an environment and allow for fast lightweight context switches. During run-time, the global task queue is processed sequentially from its beginning to its end by dequeuing one or multiple tasks by each UE. We assume, each UE is able to process each task and that all tasks are independent. Figure 3 illustrates the query execution with QTM-DLB. At first, a UE dequeues a task from the head of the global task queue. After that, a task dequeues as many tuples as specified by its buffer size from an input queue, applies its operator sequence to each tuple, and enqueues qualifying tuples into a output queue. The distribution of tasks among UEs can be applied either statically or dynamically in a general task model. In QTM-DLB, UEs acquire tasks dynamically on their own if computing capacities are available.

The execution of tasks in a general task model requires that UEs are scheduled by PEs for execution. In QTM-DLB, this mapping differs for different DBMS. A DBMS running on a single machine may refer to one processor as one PE. In contrast, a distributed DBMS may refer to one physical machine as one PE. In this paper, we focus on query execution on a single multi-core machine.

QTM and QTM-DLB are general enough such that all database scheduling strategies and chunk sizes shown in Figure 1 can be expressed in QTM and executed in QTM-DLB. We express different query execution strategies and chunk sizes with different task configurations, processing strategies and placement strategies. Since QTM-DLB is based on a dynamic load balancing approach, it omits a run-time scheduler. Instead, QTM-DLB lays out a schedule during compile-time that is flexible enough to adapt itself for different run-time conditions. The actual schedule is determined by the dynamic run-time behavior of processors that acquire new work (tasks) on their own if computing capacities are available. In contrast, the volcano execution model also omits a run-time scheduler but its scheduling is static and

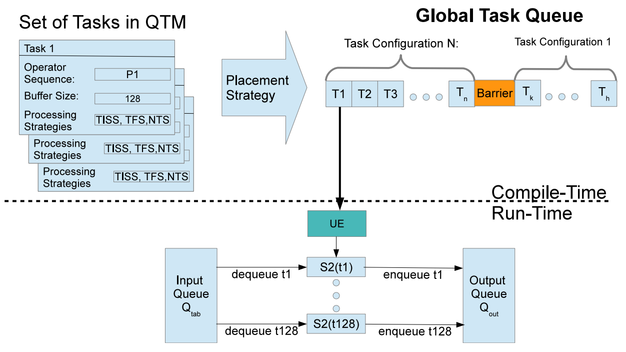


Figure 3: Query execution with QTM-DLB.

implicitly determined by its execution model. For the rest of this paper, we refer to QTM as our model that specifies query execution and *QTM-DLB* as a dynamic load balancing approach implementing QTM for query execution. In the following sections, we define QTM formally.

3.3 Task Configuration

In QTM, we define a task configuration (TC) that groups operators of a QEP. A task configuration TC_m is instantiated into n instances $\langle task_0^m \dots task_{n-1}^m \rangle$. For the rest of this paper, we refer to instance i of a task configuration TC_m as $task_i^m$. Each TC specifies a buffer B of size b in tuples and an operator sequence O_l with operators $\langle o_0^l \dots o_{n-1}^l \rangle$ for its tasks. The operators in O_l satisfy a particular order. Each tuple t_i has to be processed by each operator $\langle o_0^l \dots o_{n-1}^l \rangle$ following the order of O_l. If tuple t_i has been deleted by operator o_i, then t_i will not be processed by the remaining operator sequence $\langle o_{i+1} \dots o_{n-1} \rangle$. Additionally, we define three processing strategies NTS, $TISS$, and TFS for a TC that specify run-time execution for its tasks (see Section 3.4). The number of instances per TC is defined by $\lceil \frac{number\ of\ input\ tuples}{buffer\ size} \rceil$. Each task is self-contained and includes all information necessary to execute the operator sequence for each tuple in its buffer.

3.4 Processing Strategies

In QTM, we define three processing strategies which specify run-time execution of tasks. All tasks of the same TC share the same *new tuple* strategy NTS, *task internal scheduling* strategy $TISS$, and a *tuple fetch* strategy TFS. In the following, we present three QEP properties that require the definition of these processing strategies.

First, relational operators might create multiple output tuples from one input tuple. Thus, we define a *new tuple* strategy (NTS) for each TC. Following Manegold et al. [17], we employ two strategies for handling new tuples. With NTS_{keep}, we refer to a strategy that keeps newly created tuples of operator o_i inside a task by adding them to its buffer. Thus, new tuples are processed by the following operator sequence $\langle o_{i+1} \dots o_{n-1} \rangle$. With NTS_{enq}, we refer to a strategy that creates new tasks for newly created tuples. Therefore, new tasks are inserted into the global task queue after the last task of the current TC. After that, the original task processes the remaining operator sequence. With NTS_{keep}, new tuples are kept on the same PE but the amount of work per task increases; thus, introducing an imbalanced task workload. On the other hand, with NTS_{enq}, the amount of work per task remains almost constant. However, newly created tasks are probably executed by another processor; thus, reducing data locality.

Second, if an operators sequence consists of more than one operator, different execution orders of tuples/operators are possible inside a task. Thus, we define a *task internal scheduling* strategy (TISS) for each TC. With $TISS_{op}$, task internal scheduling follows an *operator-at-a-time* approach such that all tuples $\langle t_0 \dots t_{n-1} \rangle$ are processed by operator o_i before the next operator o_{i+1} is applied. Using $TISS_{op}$, a TC processing a pipeline of c operators instantiates $c * \lceil \frac{number\ of\ input\ tuples}{buffer\ size} \rceil$ tasks with $c-1$ materializations between operators. With $TISS_{buf}$, task internal scheduling follows a *buffer-at-a-time* approach such that each tasks processes a chunk of tuples $\langle t_0 \dots t_{B-1} \rangle$ by each operator $\langle o_0 \dots o_{n-1} \rangle$. Using $TISS_{buf}$, a TC processing a pipeline of c operators instantiates $\lceil \frac{number\ of\ input\ tuples}{buffer\ size} \rceil$ tasks, each processing the entire pipeline. We do not model partial operator sequences inside tasks. If required, we would create different TC for each partial operator sequence.

Third, tuples inside a buffer can be accessed using different access patterns. Thus, we define a *tuple fetch* strategy (TFS) for each TC. With TFS_{seq}, we refer to a strategy that fetches tuples sequentially inside each task. Thus, each operator o_i accesses tuples in a sequential order $\langle t_0 \dots t_{B-1} \rangle$. With TFS_{zig}, we refer to a strategy that fetches tuples using a zig-zag access pattern. Thus, operator o_i accesses tuples in forward direction $\langle t_0 \dots t_{B-1} \rangle$ but operator o_{i+1} accesses tuples in backward direction $\langle t_{B-1} \dots t_0 \rangle$. Thus, TFS_{zig} might increase data locality for large data sets.

3.5 Queues

In QTM-DLB, we define a global task queue Q_{task} as a list of n tasks $\langle task_0 \dots task_{n-1} \rangle$ in a particular order. We refer to Q_{head} as the first element in Q_{task}; thus, the task that will be dequeued next. We refer to Q_{tail} as the last element in Q_{task}; thus, a new task will be enqueued at position Q_{tail+1}. During run-time, tasks are processed sequentially from Q_{head} to Q_{tail}.

We define three operations on Q_{task}. First, enq_{batch} inserts a batch of tasks $\langle task_0 \dots task_{n-1} \rangle$ into the Q_{task} following a placement strategy PS. Each $task_i$ is appended at Q_{tail+1}. This enqueue operation is used during compile-time. Second, $enq_{(task_i, pos)}$ inserts a single $task_i$ at position pos into Q_{task}. For example, NTS_{enq} requires these enqueue operation to insert newly created tasks into Q_{task} during run-time. Third, $dequeue_{num}$ dequeues num tasks starting from Q_{head}.

In QTM-DLB, we have to satisfy the constraints introduced by a QEP. Thus, a synchronization point is required if TC_{m+1} depends on TC_m, i. e., all tasks of TC_m have to be processed before the first task of TC_{m+1} start processing. Therefore, we define a barrier bar for Q_{task}. A barrier guarantees, that all tasks $\langle task_0^m \dots task_{n-1}^m \rangle$ of TC_m finish processing before a $task_i^{m+1}$ of TC_{m+1} starts.

Finally, we define three different data queues. Each input relation is modeled as a *table queue* Q_{tab}. Each table queue consists of n tuples $\langle t_0 \dots t_{n-1} \rangle$. Tuples in Q_{tab} are dequeued buffer-wise depending on the buffer size of the accessing task. Q_{int} defines an *intermediate* data queue for materializations. Note, each blocking operator and each barrier requires an implicit materialization of its result. With Q_{out}, we refer to a global output queue that stores the query result.

3.6 Parallelism in QTM

With QTM, we are able to express three forms of parallelism [9]. First, *partitioned parallelism* can be exploited by partitioning the input of an operator such that all partitions can be processed in parallel (intra-operator parallelism). In QTM, we model one TC for each partitionable operator and instantiate one task for each partition. Second, *pipelined parallelism* can be exploited by processing the entire pipeline without interruption or materialization. Note, operators in a pipeline are non-blocking and do not interfere with each other (inter-operator parallelism). In QTM, we model one TC containing the entire pipeline as a operator sequence. Additionally, we apply $TISS_{buf}$ for task internal scheduling. Third, *independent parallelism* can be exploited by executing independent pipelines in parallel (inter-operator parallelism). In QTM-DLB, we support independent parallelism by placing tasks from independent pipelines interleaved into the global task queue.

We optimize parallel query execution in QTM-DLB by four means. First, we improve *temporal locality* by grouping tuples into buffers and pipelines into operator sequences for tasks. Thus, we increase the probability for tuples to reside in cache for their entire processing. Furthermore, by processing tuples in chunks, we amortize the overhead per operator call through many tuples [20, 4]. Second, we improve *spatial locality* by accessing tuples sequentially inside a buffer. The sequential access pattern leads to an increased cache line utilization and efficient prefetching. Third, we achieve a high *degree of parallelism* by specifying independent tasks that allow for asynchronous processing. Thus, independent tasks mitigate dependencies and reduces synchronization overhead. Fourth, we achieve high *resource utilization* by a loosely coupling of processing units and tasks.

4. QUERY EXECUTION SCHEDULES

In this section, we model common database schedules with QTM. Using a simple QEP, we demonstrate how common DBMS would implement different schedules. We show, that schedules mainly differ by their buffer size (chunk size) and their applied task internal scheduling strategy (TISS). For the following considerations, we assume one TC processing a pipeline of n operators. We omit NTS and TFS because they can be applied to any schedule. Figure 4 shows the schedules presented in this section. The buffer size refers to the number of tuples each task processes in a particular schedule. The operator count specifies the number of operators in the operator sequence of each task. For example, a task following a *tuple-at-a-time* + $TISS_{buf}$ schedule would process the entire pipeline with one tuple. In contrast, a task following a *tuple-at-a-time* + $TISS_{op}$ schedule would process only one operator with one tuple.

A *tuple-at-a-time* schedule performs one operator call for each tuple. The *volcano execution model* is one common example implementing this schedule [10]. We model a tuple-at-a-time schedule by defining a buffer size of one. To support row and column-oriented storage layouts, we utilize a `fetch` function for each operator call to fetch the next tuple t_{i+1}. The actual implementation of the `fetch` function differs depending on the storage layout. For a row-oriented storage layout (NSM), one memory access returns the entire tuple. For a column-oriented storage layout (DSM), the function has to collect all required attributes from v columns; thus,

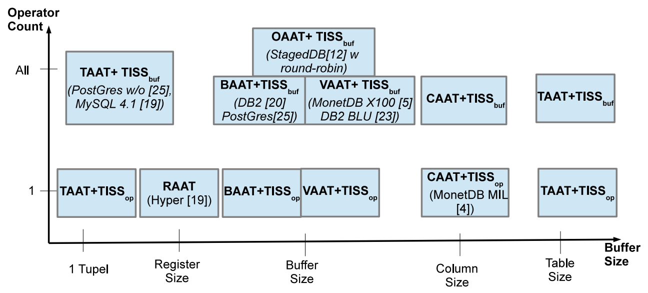

Operator Count
All
1
TAAT+ TISS_buf (PostGres w/o [25], MySQL 4.1 [19])
OAAT+ TISS_buf (StagedDB[12] w round-robin)
BAAT+TISS_buf (DB2 [20] PostGres[25])
VAAT+ TISS_buf (MonetDB X100 [5] DB2 BLU [23])
CAAT+TISS_buf
TAAT+TISS_buf
TAAT+TISS_op
RAAT (Hyper [19])
BAAT+TISS_op
VAAT+TISS_op
CAAT+TISS_op (MonetDB MIL [4])
TAAT+TISS_op
1 Tupel
Register Size
Buffer Size
Column Size
Table Size
Buffer Size

Figure 4: Query Execution Schedules in QTM.

resulting in v memory accesses. Considering performance, one operator call per tuple results in a large overhead due to many operator calls. We identify two possible operator sequences. With $TISS_{buf}$, a task processes one tuple t_i by all operators $\langle o_0 \dots o_{n-1} \rangle$. With $TISS_{op}$, each task executes one operator o_i for one tuple t_i. However, o_i has to be processed entirely for all tuples $\langle t_0 \dots t_{n-1} \rangle$ before o_{o+1} starts processing. Thus, with $TISS_{op}$, operators are processed in a step-wise manner which requires materialization of intermediate results. In contrast, with $TISS_{buf}$, tuples are only materialized while percolating the pipeline.

A *register-at-a-time* schedule was introduced by Neumann and implemented in Hyper [19]. This schedule combines operators inside the same pipeline into one operator. The *combined* operator processes as many tuples as fit into one CPU register. Therefore, the buffer size depends on the size of a CPU register and the size of a tuple. The combined operator reduces the number of operator calls to one call per pipeline per buffer. Therefore, the overhead per operator call is amortized over all tuples in the buffer and over all operators in the pipeline. Since the pipeline is compressed to only one operator per pipeline, only one possible execution order exists. Thus, we omit $TISS$ in Figure 4. Although Neumann [19] evaluates this approach for DSM, it would also be applicable to NSM.

A *buffer-at-a-time* schedule performs one operator call for each buffer. In general, the buffer can be of any size. However, previous work shows that a buffer size that matches a hardware parameter exhibits the best performance [28, 20, 7, 25]. Common examples are the size of the L1, L2 or L3 cache. DB2 5.2 [20] as well as PostgreSQL 7.3.4 [25] implement this buffer-at-a-time schedule. In addition to these static buffer sizes, Cieslewicz et al. [8] introduce a buffer that changes its size dynamically based on cache miss sampling. To take the content of a buffer into account, we restrict the buffer-at-a-time schedule to a NSM storage layout. A buffer storing tuples of a NSM storage layout consists of the entire tuple with all attributes. In contrast, a buffer storing tuples of a DSM storage layout usually consists of attribute values for a single column. Thus, we model the buffer-at-a-time schedule only for the NSM storage layout and leave the DSM storage layout for the vector-at-a-time schedule. The operator sequences are similar to the tuple-at-a-time schedule. However, instead of processing one tuple, a task following $TISS_{buf}$ processes all tuples in its buffer B at operator o_i before processing the same buffer at the next operator o_{i+1}. With $TISS_{op}$, a task processes one buffer B with one operator o_i. However, o_i has to be processed entirely for all buffers $\langle B_0 \dots B_{n-1} \rangle$ before o_{o+1} starts processing and thus materialization is required. Considering performance, the overhead per operator call for $TISS_{buf}$ and $TISS_{op}$ is amortized over all tuples in the buffer. Thus, the advantages of the block-oriented processing [23, 20] are exploited. Additionally, tasks following $TISS_{buf}$ amortize their overheads over all operators in the pipeline.

A *vector-at-a-time* schedule performs one operator call for each vector of each column. DBMS implementing this schedule are MonetDB/X100, C-Store, and DB2 with BLU. MonetDB/X100 [5] and DB2 with BLU [23] attempt to hold all data in caches, C-Store [24] processes blocks of 64KB. The vector-at-a-time schedule is essentially a buffer-at-a-time schedule but introduces one buffer per column. In contrast, a buffer-at-a-time schedule introduces one buffer for the entire relation or between operators. Additionally, a buffer-at-a-time schedule determines the buffer size in relation to the size of an entire tuple. In contrast, a vector-at-a-time schedule has to determine a separate buffer size for each column in relation to the size of the attribute values. The number of buffers increases with each additional column. One major advantage of a vector-at-a-time schedule is its opportunity for *vectorized processing.* Vectorized processing allows for the usage of *Single Instruction Multiple Data (SIMD)* that showed an improved performance [28, 5]. Another important advantage of a vector-at-a-time schedule is its increased buffer utilization if only a small fraction of all attributes are accessed. In contrast, a buffer-at-a-time schedule on a NSM storage layout would load unused data into its buffer if only a small fraction of all attributes are accessed. The operator sequences are similar to the buffer-at-a-time schedule but extend one call per buffer to one call per buffer per column. The processing of $TISS_{buf}$ and $TISS_{op}$ inside the operator sequences remains unchanged but an additional call for each columns is added. Note, the processing of different columns per operator introduces an opportunity for scheduling columns in different orders.

A *column-at-a-time* schedule performs one operator call for each column. MonetDB/MIL [4] implements this schedule. It requires a DSM storage layout and corresponds to a vector-at-a-time approach with the entire columns as one vector. However, when executing a column-at-a-time schedule using multiple PEs, the entire column can be partitioned into chunks, i. e., this schedule transforms into a vector-at-a-time schedule. Processing a column entirely introduce additional costs for materialization of intermediate results; thus, increases the memory consumption [4]. The buffer size corresponds to the number of tuples in a column. With $TISS_{buf}$, a task processes one column col_i entirely with operator o_i before processing the same column with the next operator o_{i+1} With $TISS_{op}$, a task processes col_i by operator o_i and all columns $\langle col_0 \ldots col_{n-1} \rangle$ have to be processed by o_i before o_{o+1} starts processing.

A *table-at-a-time* schedule performs one operator call for the entire table and can be found in OLTP databases that apply a *data manipulation operation* to an entire table. To support a row-oriented and column-oriented storage layout, we utilize the `fetch` function introduced for a tuple-at-a-time schedule. When executing a table-at-a-time schedule using multiple PEs, the entire table can be partitioned into chunks, i. e., this schedule transforms into a buffer-at-a-time schedule. The processing with $TISS$ are similar to a buffer-at-a-time schedule with one buffer for the entire table.

An *operator-at-a-time* schedule represents a special schedule that follows the *StagedDB* approach. This schedule is implemented in STEPS and QPipe [12]. An operator sequence is divided into stages that represent operators. The buffer size corresponds to the size of an input queue at each stage. The stages exchange tuples via messages from one input queue to another. While not stated, we assume STEPS and QPipe work on a NSM storage layout because the prototypes are based on Shore and BerkeleyDB which use a NSM storage layout [12]. The actual operator sequence depends on the applied scheduling algorithm. A simple round-robin scheduling will call each operator for a fix time slice before calling the next in a circular manner. However, due to back pressure or other scheduling decisions, an arbitrary operator sequence is possible. Based on the scheduling algorithm, each stage processes tuples in its input queue as long as its time slice is valid or until its input queue becomes empty.

5. EVALUATION

In this section, we evaluate different schedules that are modeled in QTM. At first, we describe our experimental setup in Section 5.1. After that, we introduce our test schedules in Section 5.2. Then, we compare them with respect to run-time in Section 5.3 and resource utilization in Section 5.4. Finally, we examine their scalability Section 5.5.

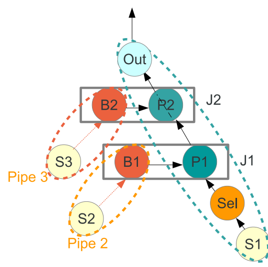
5.1 Experimental Setup

5.1.1 Prototype

We implement QTM-DLB as a prototype in C++. QTM-DLB executes queries modeled in QTM (see Section 3.2). In a preparation step, we create a set of tasks and place them into a global task queue. The order of tasks and their configuration represent a schedule. In this paper, we exclude the preparation step for our measurements and measure solely query execution during run-time. Query execution in QTM-DLB proceeds as follows. At first, each processor dequeues a tasks from the global task queue. After that, each task dequeues all tuples for its processing from an input or intermediate queue into its buffer and applies its operator sequence to each tuple. The processing strategies of each task specify the execution order of operators, the access pattern for tuples in its buffer, and the processing of newly created tuples. Finally, each task enqueues qualifying tuples into a global output or intermediate queue. This sequence is repeated until all tasks in the global task queue are processed.

Although tasks run asynchronously and mainly process *task-local* data in their buffers, they have to synchronize on shared data structures. In our prototype, we have to synchronize 1) dequeuing of tasks from the global task queue, 2) dequeuing of tuples from an input or intermediate queue, and 3) enqueuing of result tuples into a intermediate or global output queue. In QTM-DLB, we synchronize these three queue operations with atomic counters as proposed by Cieslewicz et al. [7]. Thus, each enqueuing or dequeuing operation of n tasks or tuples increments an atomic counter by n. After that, a task can exclusively access tasks or tuples from n_{old} to n_{new-1}. Note, these three synchronization operations represent the overhead introduced for each task in QTM-DLB.

Within each task, bookkeeping of qualified tuples among different operators is maintained by a bitmask. Bit i in a bitmask represents the qualification of tuple i in buffer B. Each operator applies it processing only if tuple i was qualified by previous operators. Then, each operator updates the bitmask using an `AND` operation for its qualified tuples.



	Workload
Tuples per Relation	30M
Selection	< 25M
S1 Values	0,1,2,...
S2 Values	0,2,4,...
S3 Values	0,4,8,...

Figure 5: TestCase: Multi-Level Join.

Finally, the last operator in a pipeline places all qualified tuples into the global output queue. We leave tuple modification inside a pipeline, e. g., a concatenation of two attribute values, for future work.

In QTM-DLB, we implement a selection operator and a hash join. Each tuple in an input relation consists of an 8 byte key and an 8 byte payload. The hash join is implemented as a non-partitioning hash join following Blanas et al. [3] with the improvement of an contiguous array for buckets proposed by Balkesen et al. [2]. Each hash table consists of small buckets with 32 entries per bucket. Each bucket entry consists of an 8 byte key and an 8 byte pointer. We implement the same the hash function as applied in *PostgreSQL*[1].

5.1.2 *Workload*

In our evaluation, we model different schedules for the QEP shown in Figure 5. The QEP consists of three input relations, one selection operator, and two hash based equi-joins. The dataset is synthetically generated and consists of three relations containing 30M tuples in ascending order. We introduce skew by incrementing tuples with different values as shown in Figure 5. As a result, each join has a selectivity of 0.5. Furthermore, the selection at the beginning of the pipeline filters 5M tuples.

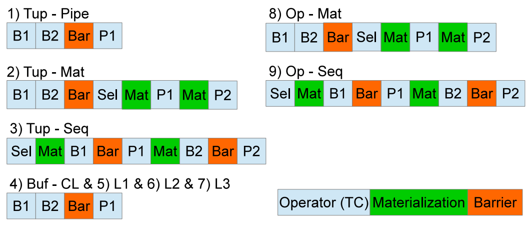
5.1.3 *Hardware and Software*

We evaluate our prototype on an Intel Xeon E7-4870 CPU. The CPU contains of 10 physical cores, each supporting *hyper-threading*. The cache hierarchy of each core is composed of a separate 32KB L1 cache for instructions and data, each 8-way set associative. Additionally, each core owns a unified 256KB L2 cache for data and instructions, each 8-way set associative. The L1 and L2 cache are exclusive to each core. Finally, all cores share a 32MB 24-way set associative L3 cache. We ran our experiments on an *openSUSE 13.1* using a 3.14.4 kernel. Our prototype was compiled with GCC 4.8.1 using O3 compiler optimizations. We measure performance counters using the PAPI framework[2].

5.2 Test Schedules

For our evaluation, we implemented nine different schedules for the QEP shown in Figure 5. In Figure 6a, we illustrate these schedules as a sequence of operators. We model one operator as one TC and instantiate tasks as shown in Figure 6b. In general, the buffer size determines the number of tasks per operator and is either fixed (Schedule 1-3), matches a cache size (Schedule 4-6), or is determined

[1]http://www.postgresql.org/
[2]http://icl.cs.utk.edu/papi/


1) Tup - Pipe
B1 B2 Bar P1
2) Tup - Mat
B1 B2 Bar Sel Mat P1 Mat P2
3) Tup - Seq
Sel Mat B1 Bar P1 Mat B2 Bar P2
4) Buf - CL & 5) L1 & 6) L2 & 7) L3
B1 B2 Bar P1
8) Op - Mat
B1 B2 Bar Sel Mat P1 Mat P2
9) Op - Seq
Sel Mat B1 Bar P1 Mat B2 Bar P2
Operator (TC)
Materialization
Barrier

(a) Test Schedules.

X-At-A-Time Approaches	Schduling Type	Task-internal Scheduling (TISS)	Buffer Size in Tuples	Tasks per Op	Total Tasks
1) Tup - Pipe	T-AAT	TISS(buf)	1	30M	90 M
2) Tup - Mat	T-AAT	TISS(op)	1	30M	150 M
3) Tup - Seq	T-AAT	TISS(op)	1	30 M	150 M
4) Buf - CL	B-AAT	TISS(buf)	4	7.5 M	22,5 M
5) Buf - L1	B-AAT	TISS(buf)	2048	14.649	43.947
6) Buf - L2	B-AAT	TISS(buf)	16384	1.832	5.496
7) Buf - L3	B-AAT	TISS(buf)	491.520	62	186
8) Op - Mat	O-AAT	TISS(op)	7.5 M	4	20
9) Op - Seq	O-AAT	TISS(op)	7.5 M	4	20

(b) Test Configurations. (dop=4)

Figure 6: Test Cases.

in relation to the current dop (Schedule 7-9). In contrast, the scheduling strategy determines the number and order of operators as well as the number of materializations and barriers.

We model Schedules 1-3 in QTM with different *task-internal scheduling strategies* ($TISS$) as *tuple-at-a-time* schedules (T-AAT). Schedules 1-3 represent three possible schedules for the volcano execution model using a buffer size of one. Since each operator instantiates one task per tuple, 30 million tasks per operator are created. However, the total number of operators differ between Schedule 1-3 due to different execution orders. Tasks in Schedule 1 and 2 build hash tables $B1$ and $B2$ for relation $S2$ and $S3$ until the barrier is reached. The barrier satisfies the constraint that the first probe operator has to wait until all hash tables are built entirely. We model Schedule 1 in QTM with $TISS_{buf}$. Thus, each task processes the entire pipeline for one tuple. In contrast, we model Schedules 2 and 3 in QTM with $TISS_{op}$. Thus, all tasks cooperatively finish the processing of one operator and materialize their results before processing the next operator. Note, materialization eliminates pipeline parallelism and increases the number of tasks due to an increased number of TCs. As shown in Figure 6a, we combine a table scan and a hash build into one operator $B1$ or $B2$. Thus, $Pipe2$ and $Pipe2$ are reduced to one operator. A pipeline containing only one operator allows only one possible execution order. Thus, $TISS$ did not affect execution order of $B1$ or $B2$.

In contrast to Schedule 1 and 2, we model Schedule 3 in QTM with $TISS_{op}$ as a schedule executing a sequential join order; thus, joins are not interleaved. The execution order changes to 1) applying the selection to each tuple in $S1$, 2) building the hash table for $S2$ and probing the intermediate result of the selection ($S1$) in $B1$, and 3) building the hash table for $S3$ and probing the intermediate result of the previous probe ($P1$) in $B2$. As shown in Figure 6a, the sequential join order changes the execution order and increases the number of barriers. However, the total number of tasks remains equal to Schedule 2.

We model Schedules 4-7 in QTM with $TISS_{buf}$ as *buffer-at-a-time* schedules (B-AAT). The buffer sizes match differ-

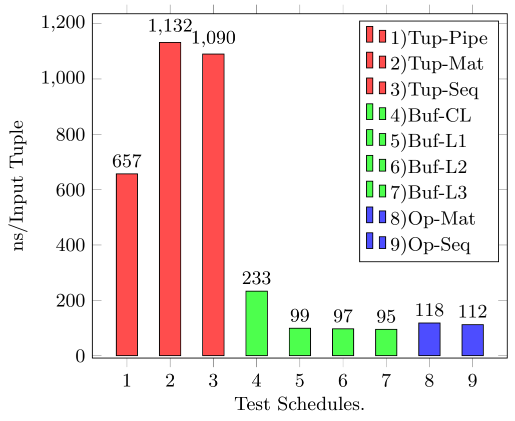


Figure 7: Run-times for Test Schedules (dop = 4).

ent cache sizes. Schedules 4-6 determine their buffer size such that all tuples fit into a cache line (Schedule 4), L1 cache (Schedule 5), or L2 cache (Schedule 6). Schedule 7 divides the L3 cache between the number of executing threads (dop). Thus, the buffer size is determined by $\lceil \frac{size\ of\ L3}{dop} \rceil$. Similar to Schedule 1, Schedule 4-7 exploit pipeline parallelism but execute *Pipe*1 for a chunk of tuples. As shown in Figure 6b, an increased buffer size reduces the number of tasks per operator. The total number of tasks ranges from 186 to 22,5M. Note, Schedules 4-7 are common in the volcano execution model using block-oriented processing or in a cache-conscious run-time scheduler.

We model Schedules 8 and 9 in QTM with $TISS_{buf}$ as *operator-at-a-time* (O-AAT) schedules. The buffer size is determined by dividing the input tuples equally between available threads. Thus, the number of tasks is equal to the number of threads (dop). Schedules 8 and 2 as well as 9 and 3 model the same execution order and number of operators. However, the buffer sizes differ significantly. Schedules 2 and 3 model the smallest possible buffer size of one tuple and Schedule 8 and 9 model the largest buffer with regard to dop. These different buffer sizes impact the number of tasks significantly. Schedules 8 and 9 might be found in MonetDB [4].

5.3 Run-time

In Figure 7, we show the run-time in nanoseconds per input tuple for each test schedule presented in Section 5.2. We execute each schedule in QTM-DLB with a dop of 4. Figure 7 shows, that B-AAT Schedules 5-7 achieve the shortest run-times and T-AAT Schedules 1-3 the longest. Furthermore, O-AAT Schedules 8 and 9 are slightly slower that the B-AAT Schedules 5-7 but faster than B-AAT Schedule 4.

Schedule 1 implements the most efficient T-AAT schedule that reduces the run-time by a factor of almost two compared to Schedules 2 and 3. The main reasons are 1) the reduced number of tasks and 2) the exploitation of pipeline parallelism. Schedules 2 and 3 execute the same number of tasks, materialize the same number of intermediate results, and do not exploit pipeline parallelism. However, they model a different execution order that shows only marginal impact on the run-time.

B-AAT Schedules 4-7 model differently sized buffers which result in different numbers of tasks. Schedule 4 executes 22,5M tasks, each loading as many tuples as fit into a cache line (4 tuples). Although, Schedule 4 decreases the run-time by a factor of 5 compared to Schedules 2 and 3, the large number of tasks results in the longest run-time of all B-AAT schedules. Schedules 5-7 decrease the run-time by a factor of 2 compared to Schedule 4 and by a factor of 10 compared to Schedules 2 and 3. However, the run-times for Schedule 5-7 are very similar. Schedule 5 executes about 40K tasks and performs slightly worse than Schedule 6 executing roughly 5K tasks. Schedule 7 achieves the best overall run-time by executing only 186 tasks.

O-AAT Schedules 8 and 9 execute fewer tasks than all other schedules (only 20 tasks). As in Schedules 1-3, the impact of a different execution order on the run-time is only marginal. However, Schedules 8 and 9 with very large buffers improve run-time by a factor of 10 compared to Schedules 2 and 3 executing the same schedule with very small buffers. In the next section, we present explanations for these different run-times by sampling the query execution.

5.4 Time Distribution

We analyzed the utilization of the cache hierarchy by our test schedules to explain the different run-times. We show, that the cache hierarchy impacts the run-times to a high degree. Figure 8 and Figure 9 summarize our sampling results. Additionally, Figure 10 shows the breakdown of misses. In these figures, a counter samples either data cache misses (DCM), instruction cache misses (ICM), or misses in a unified instruction and data cache (CM). Additionally, we measure *Translation Lookaside Buffer* misses for data pages (TLB DM) and instruction pages (TLB IM) as well as the number of branch miss predictions (Branch MP).

5.4.1 Observations

As a first observation, Schedules 2 and 3 as well as 8 and 9 reflect similar counter values in Figure 8 and Figure 9. These similar numbers of cache and TLB misses explain similar run-times observed in Figure 7. However, the sequential join execution of Schedules 3 and 9 causes slightly less misses and thus improves the run-time marginally.

As a second observation, the improved run-time of Schedule 1 compared to Schedules 2 and 3 can be attributed to less data and instruction cache misses. The main reasons for that are threefold. First, Schedule 1 exploits pipeline parallelism which reduces the number of data cache misses (L1 DCM & L2 DCM). Second, Schedule 1 executes less tasks which reduces the number of instructions and instruction related cache misses (L1 ICM & L2 ICM). Third, following the improved data and instruction cache utilization, both TLB cache misses (TLB DM & TLB IM) are reduced as well as the number of L3 cache misses (L3 CM) and branch mispredictions (Branch MP).

As a third observation, data cache misses of Schedules 8 and 9 are similar to Schedule 5. The small number of four tasks per operator for Schedule 8 and 9 results in the fewest instruction cache misses. However, the elimination of pipeline parallelism and the required materialization result in longer run-times compared to other B-AAT schedules.

The fourth observation is contrary to the general assumption that a buffer that fits entirely into a private cache exhibit less data cache misses [28, 20, 7, 25]. As shown in Figure 8, this assumption does not hold for Schedules 1 and 4 using very small buffer sizes. Besides the huge number of tasks, the reasons for that are twofold. First, Schedule 1 cannot exploit spatial locality because one cache line is shared among different tasks. Thus, each cache line is loaded multiple times. Second, a small buffer size prevents efficient prefetching. For example, if Schedule 4 is executed by n threads, each tasks loads every n-th cache line (omitting the dynamic run-time behavior). In contrast, a task in Schedule 5 accesses 512 cache lines sequentially. Thus, a hardware prefetcher may detect the access pattern of Schedule 5 but not the access pattern of Schedule 4.

Finally, the miss breakdown in Figure 10 reveals a different distribution among schedules. Note, some misses are hidden because of their marginal occurrence. The observations are fivefold. First, L2 DCM are more frequent than the other misses. Second, TLB DM and L2 DCM are almost constant over all schedules. Third, L2 ICM and TLB IM are negligible. Fourth, L3 CM are more frequent and Branch MP are less frequent for larger buffer sizes (Schedules 4-9). Fifth, L1 ICM are more frequent for smaller buffer sizes (Schedules 1-4). Note, a time breakdown can be derived from Figure 10 by multiplying the number of cache misses with actual miss penalty. In the next sections, we analyze the cache characteristics of the B-AAT Schedules 4-7 in more detail.

5.4.2 Data Cache Misses

Our sampling results for Schedules 4-7 in Figure 8 show, that data caches misses in L1, L2, and L3 cache decrease with increasing buffer size until the buffer size exceeds the largest private cache (L2). After that, data cache misses increase. The main reason for that originates from a different exploitation of pipeline parallelism. Pipeline parallelism enables data locality for tuples percolating the pipeline. In an optimal case, the first operator in a pipeline loads all tuples into the cache. Then, each consecutive operator will suffer no cache miss because its data is already loaded. The number of cache misses is reduced as long as the data set fits into the cache. Unfortunately, this optimal case requires that only the first operator in a pipeline loads the entire data set. However, this requirement is usually not satisfied because consecutive operators like a hash probe may also load data into the cache. Each additional data load increases the probability that already loaded tuples are evicted before reuse (so-called *cache-thrashing*). As shown in Figure 8, cache trashing occurs as soon as the buffer size exceeds the private L2 cache. Starting from this buffer size, cache misses increase up to a point where each data access results in a cache miss and no data locality inside the pipeline can be exploited. The TLB data cache misses follow this evolution.

5.4.3 Instruction Cache Misses

Our sampling results in Figure 9 show, that instruction cache misses are correlated with the number of tasks. Thus, schedules processing tasks with large buffers (Schedules 7-9) decrease the total number of tasks and amortize their task overhead over multiple tuples. Additionally, they increase the locality of instructions by processing multiple tuples in tight loops. The improved instruction locality results in less instruction cache misses.

Compared to data cache misses, instruction cache misses are more performance critical because they cannot be overlapped using *out-of-order* execution [12]. In the worst case, the processor pipeline stalls until the instructions are fetched.

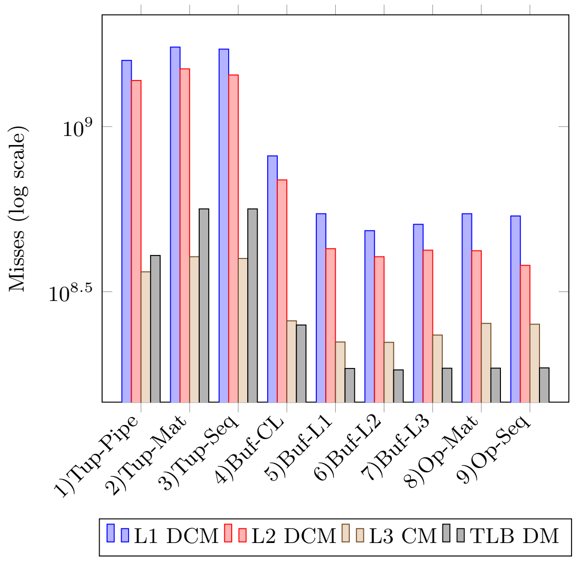


Figure 8: Data Related Misses.

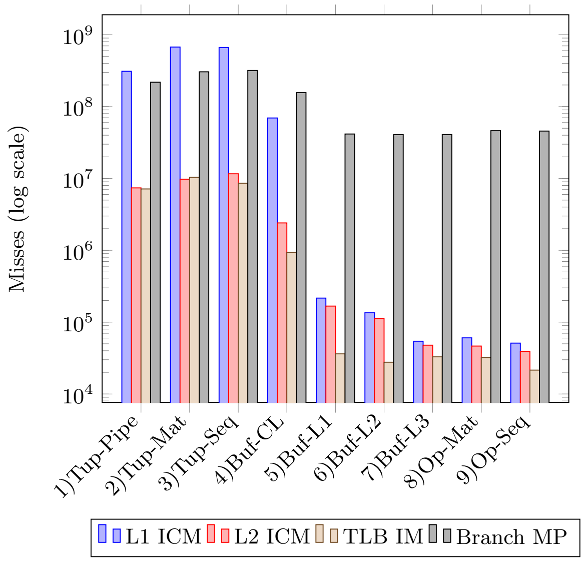


Figure 9: Instruction Related Misses.

This difference is the main reason why Schedule 7 is faster than Schedule 6. Although Schedule 7 suffers more data cache misses, the reduced number of performance critical instruction cache misses are more crucial for the run-time. TLB instruction misses and branch miss predictions follow the characteristics of the instruction cache. Furthermore, small tasks result in more branch mispredictions because the number of branch targets are increased which pollutes the *branch target buffer*.

5.4.4 Results

To sum up, we identified a trade-off between data and instruction cache performance. We show, that a schedule that is optimized for data cache locality does not necessarily outperform a schedule optimized for instruction cache locality. Overall, the cache performance can be adjusted by the buffer size which impacts data cache as well as instruction cache performance. We show, that a schedule that produces medium sized tasks (Schedules 4-6) by determining its buffer size based on cache size exploits data locality efficiently. On

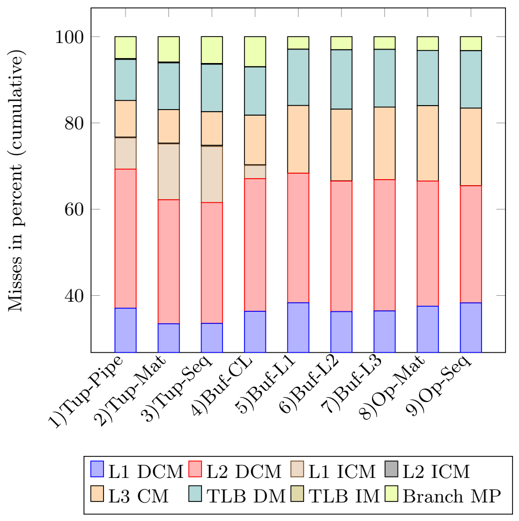


Figure 10: Breakdown of Misses.

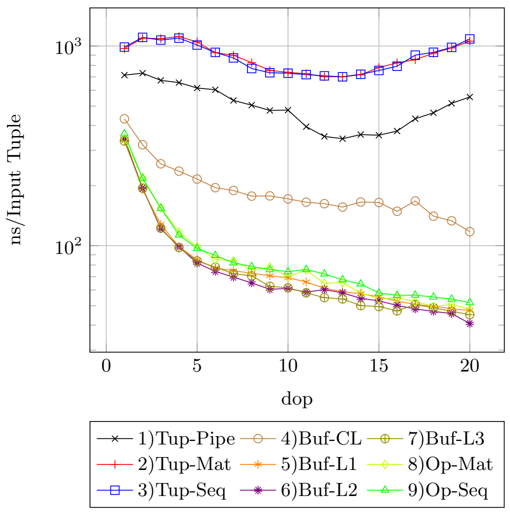


Figure 11: Scalability

the other hand, the high number of tasks introduce many instructions; thus, causing many instruction cache misses. In contrast, a schedule that produces large tasks (Schedules 7-9) cause less instruction cache misses due to a decreased number of instructions and exploits instruction locality efficiently. On the other hand, few large tasks cause more data cache misses if the buffer size exceeds the data cache size.

5.5 Scalability

In Figure 11, we examine the scalability of our test schedules. Starting at a dop of 10, hyper-threading is applied. In general, hyper-threading interleaves threads at a fine granularity and is beneficial if two threads execute different types of work. For example, if one thread handles an *I/O* request and the other executes computation [26]. Although hyper-threading introduces two logical cores per physical core, both cores have to share many execution resources, including memory bus and caches [26].

T-AAT Schedules 2 and 3 scale up to a dop of 11, then stagnate between 12-14 before increasing run-time starting from 15. However, the best reported speedups of nearly 1.4 with 13 cores for both schedules are only marginal. Even worse, with 20 cores, Schedules 2 and 3 are nearly as fast as running the same schedule with only two cores. Schedule 1 scales with the same characteristics but exhibits a slightly larger speedup of two. The reasons for the poor scalability of the T-AAT schedules are threefold. First, Schedules 1-3 cannot produce enough independent work to overlap data cache misses with useful computation. Schedule 1 scales slightly better because it exhibits less cache misses and thus free up memory bandwidth that is available for other cores. However, Schedules 1-3 are *memory-bound*. Second, threads execute tasks that perform similar work on different data. If executing two threads on the same core using hyper-threading, both threads require the same execution units and thus the benefit of hyper-threading cannot be exploited to its full extent. Furthermore, the available resources per thread are divided by two and threads may evict tuples mutually. Third, context switches between threads are less expensive with hyper-threading but still introduce some overhead.

B-AAT Schedules 4-7 scale much better and do not increase run-time if hyper-threading is applied. The main reason for that are the more efficient exploitation of the cache hierarchy. Therefore, more data accesses can be overlapped with computation and more data accesses can be satisfied by private caches. Schedule 6 achieves the highest speedup (8.3), followed by Schedule 7 (7.4), Schedule 5 (7.2), and Schedule 4 (3.6). The shortest run-time is achieved by Schedule 6 with a dop of 20. However, Schedules 6 and 7 compete for the best run-time in Figure 11. Schedule 6 achieves shorter run-times for a dop less than 10 and Schedule 7 for a dop larger than 10. Overall, Schedule 6 achieves the best run-time in 11 of 20 samples. The reason for this competition is the trade-off between data and instruction cache misses as described in Section 5.4. Therefore, Schedule 6 produces less data but more instruction cache misses and Schedule 7 produces less instruction but more data cache misses.

Schedules 8 and 9 exhibit same characteristics as Schedules 5-7 but with slightly longer run-times and less speedup of 7.1 for Schedule 8 and 6.9 for Schedule 9. The difference can be attributed to the elimination of pipeline parallelism and the materialization of intermediate results.

As a result, we identify a sweet spot where the ratio of data locality and instruction locality produces the fastest schedules. The sweet spot lies between a schedule with a buffer size matching the largest private cache (Schedule 6) and a schedule with a slightly larger buffer size that reduces instruction cache misses (Schedule 7).

6. RELATED WORK

Approaches for dynamic load balancing in research vary in the number of task queues and granularity of tasks [6, 17, 16, 22]. There are approaches using one global task queue [17], one queue per thread per operator [6], one queue per processor and one global queue [16], or one queue per processor socket [22]. The granularity of tasks also vary among different approaches. While one approach did not state how to convert a QEP into a set of tasks [6], two create tasks mainly from partitionable operators like aggregations or hash builds [16, 22]. Manegold et al. [17] use the call of one operator with one tuple as the basic granularity of one task. However, neither of these approaches considers a task granularity different from one operator call for one tuple. Additionally, locality of data and instructions inside a cache hierarchy and different execution orders are not considered. Furthermore, we extend the notion of tasks by a generalized work and data specification and a declaration of processing strategies which specify task execution during run-time.

The optimal chunk size was only examined for a particular scheduling strategy. Padmanabhan et al. introduce *block-oriented* processing that extends the volcano query execution model to process a block of tuples. Zhou and Ross [25] implement the block-oriented approach by inserting buffers between certain operators to improve the instruction cache performance. Zukowski et al. [28] compare the row-wise storage format (NSM) and column-wise storage format (DSM) in combination with the block-oriented approach. However, neither of these approaches considers the impact of different scheduling strategies nor take the exploitation of pipeline parallelism into account.

Previous work sampled commercial DBMS workloads to identify the distribution between time spent for computation and time spent for waiting for data. In our context, the OLAP workloads in [5, 11, 1] are most relevant. Ailamaki et al. [1] discovered, that on the average, half of the execution time is spent in stalls while 90% of the memory stalls are due to L2 data cache misses and L1 instruction cache misses. Furthermore, they show that databases are particularly ineffective in taking advantage of modern superscalar processor capabilities [1, 5]. Our evaluation contributes to this observation by adding the chunk size and scheduling strategy as new dimensions that impact the distribution of cache misses.

7. CONCLUSION AND FUTURE WORK

In this paper, we classified common databases by their scheduling strategy and chunk size. Furthermore, we introduced a *Query Task Model (QTM)* that allows us to express and compare different approaches for parallel query execution. With QTM, we open a design space for database schedules. In our evaluation, we examined how different schedules exploit resources of modern CPUs. We showed, that a schedule that is optimized for data cache locality does not necessarily outperform a schedule optimized for instruction cache locality. Furthermore, we identified a sweet spot where the ratio of data locality and instruction locality produces the fastest schedules. Future work will focus on the development of a general framework for transforming an arbitrary QEP into QTM. Furthermore, we will investigate work sharing among concurrent queries. Finally, we are working on a cost model that predicts the costs of different task configurations on different hardware architectures.

8. REFERENCES

[1] A. Ailamaki et al. Dbmss on a modern processor: Where does time go? In *VLDB*, 1999.

[2] C. Balkesen et al. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

[3] S. Blanas et al. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, 2011.

[4] P. Boncz et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[5] P. A. Boncz et al. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[6] L. Bouganim et al. Dynamic load balancing in hierarchical parallel database systems. In *VLDB*, 1996.

[7] J. Cieslewicz, K. A. Ross, and I. Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

[8] J. Cieslewicz et al. Cache-conscious buffering for database operators with state. In *DaMoN*, 2009.

[9] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *SIGMOD*, 1996.

[10] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, 1990.

[11] N. Hardavellas et al. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, 2007.

[12] S. Harizopoulos and A. Ailamaki. Stageddb: Designing database servers for modern hardware. *IEEE Data Eng. Bull.*, 2005.

[13] W. Hong. Exploiting inter-operation parallelism in xprs. In *SIGMOD*, 1992.

[14] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[15] V. Leis et al. Morsel-driven parallelism : A numa-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[16] H. Lu and K.-L. Tan. Dynamic and load-balanced task-oriented datbase query processing in parallel systems. In *EDBT*, 1992.

[17] S. Manegold, J. K. Obermaier, and F. Waas. Load balanced query evaluation in shared-everything environments. In *Euro-Par*, 1997.

[18] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming.* Addison-Wesley Professional, first edition, 2004.

[19] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, 2011.

[20] S. Padmanabhan et al. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

[21] H. Pirahesh et al. Parallelism in relational data base systems: Architectural issues and design approaches. In *DPDS*, 1990.

[22] I. Psaroudakis et al. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In *ADMS*, 2013.

[23] V. Raman et al. Db2 with blu acceleration: So much more than just a column store. In *VLDB*, 2013.

[24] M. Stonebraker et al. C-store: A column-oriented dbms. In *VLDB*, 2005.

[25] J. Zhou and K. A. Ross. Buffering databse operations for enhanced instruction cache performance. In *SIGMOD*, 2004.

[26] J. Zhou et al. Improving database performance on simultaneous multithreading processors. In *VLDB*, 2005.

[27] M. Ziane et al. Parallel query processing in dbs3. In *PDIS*, 1993.

[28] M. Zukowski et al. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In *DaMoN*, 2008.