On inter-operator data transfers in query processing

Harshad Deshmukh[§], Bruhathi Sundarmurthy[§], Jignesh M. Patel
{harshad, bruhathi, jignesh}@cs.wisc.edu
University of Wisconsin - Madison

***Abstract*—In designing query processing primitives, a crucial design choice is the method for data transfer between two operators in a query plan. As we were considering this critical design mechanism for an in-memory database system that we are building, we quickly realized that (surprisingly) there isn't a clear definition of this concept. Papers are full of ad hoc use of terms like pipelining and blocking, but these terms are not crisply defined, making it hard to fully understand the results attributed to these concepts. To address this limitation, we introduce a clear terminology for how to think about data transfer between operators in a query pipeline. We argue that there isn't a clear definition of pipelining and blocking, and that there is a full spectrum of techniques based on a simple concept called unit-of-transfer. Next, we develop an analytical model for inter-operator communication, and highlight the key parameters that impact performance (for in-memory database settings). Armed with this model, we then apply it to the system we are designing and highlight the insights that we gathered from this exercise. We find that the gap between the traditional "pipelining" and "non-pipelining" methods of query processing, w.r.t. key factors such as performance and memory footprint is quite narrow, and thus system designers should likely rethink the notion of "pipelining" vs. "blocking" for in-memory database systems.**

I. Introduction

A fundamental consideration in analytic query processing design is the mechanism for communicating data between operators, such as a select operator feeding to an aggregate operator, or a select operator feeding to a probe operator to evaluate a hash join. Typically the source operator is called the *producer* and the destination is called the *consumer*. There are two broad camps for *intra-operator communication* methods, in both traditional disk-based and newer in-memory systems. These two camps sharply distinguish themselves based on the data transfer method between producers and consumer. These camps are pipelining [44], [25] and blocking [23], [13].

Understanding the implication of choosing one method over the other is non-trivial since there are varied definitions of what comprises pipelining or blocking. For example, in [26], the definition of pipeline leans towards "a tuple being processed should be present in the register". Vectorwise [44] departed from the traditional tuple-at-a-time processing model and proposed hyper-pipelining query execution [7] using batches (or vectors) of tuples. On the other hand, disk-based systems [15], [19], [38] define pipelining as "tuples should be successively processed without having to be sent to the disk in between".

From these examples, we observe that the line between pipelining and blocking is fuzzy and it depends on the batch size of data transfer. The first key contribution of this paper is to highlight that there is no crisp definition of pipelining or blocking and that is a key source of confusion. It is hard to understand results either for or against these mechanisms without a crisp definition. In this paper we introduce the term *unit-of-transfer* (UoT) to clarify these mechanisms. This simple concept is graphically depicted in Figure 1.

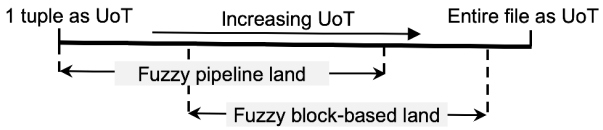


Fig. 1: **Unit of Transfer (UoT).**

With this terminology we can see that the granularity of inter-operator transfer mechanisms is really a spectrum; different systems are designed to support different UoT values. At one end of the spectrum, a tuple can be the UoT [25], whereas at the other end of the spectrum, the whole table (or the whole file or the whole intermediate result) can be the UoT [13]. Many systems such as MonetDB [23] and Quickstep [36], which produce batches of tuples as output, fall somewhere in between the two extremes.

We point out an immediate benefit of introducing the notion of UoT – it implicitly addresses the confusion about where data should reside in the memory hierarchy for the data transfer mechanism to be called 'pipelining' or otherwise. For instance, if the UoT is small, chances are that it resides in registers, or if the UoT is too big to fit in the memory, it may be forced to persistent storage. Disk-based systems try to avoid expensive disk I/O operations. Their UoT is a batch of tuples that is main-memory resident.

Next, with a new clear terminology of UoT for data transfer mechanisms, we propose a model to study the implication of changing UoT for in-memory systems. Here, we enumerate key factors that are crucial to the model and their interaction. The factors are parallelism, block size, storage format, query structure, and hardware characteristics. A combination of these factors jointly impacts the performance of a query when there is no I/O bottleneck. The collective space of combinations for these dimensions is very large, and prior work has largely looked at individual dimensions and studied their impact on overall query execution.

The analytical model helps understand the impact of these factors on query performance. It also presents system designers and practitioners with a tool to analyze the impact of UoT and other dimensions on (analytic) query processing performance.

[§]Work done while in UW - Madison. Currently employed at Alphabet.

2022 IEEE 38th International Conference on Data Engineering (ICDE) | 978-1-6654-0883-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDE53745.2022.00066

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00066

Then, as a case in point, we apply the proposed model to a specific system, Quickstep (the system background is described in Section III), and study the impact of the UoT on performance.

We speculate that the UoT model can be used as a common language to describe the previous works more clearly. Inter-operator data movement involves several implementation details. In this paper, we discuss these details in the context of the Quickstep system [36], but these insights likely also apply in other settings/systems. Our key observations are:

- We observe that the performance of queries depends on many dimensions like parallelism, block size, storage format, query plan structure and hardware characteristics like prefetching.
- We compare the memory requirements of query execution with changing UoT and observe that for TPC-H queries, the average memory overhead can be less than 4% of the base table.
- For smaller block sizes, using a smaller UoT results in higher performance compared to using a bigger UoT. As block sizes increase, the UoT does not have much impact. The latter observation is perhaps surprising given the amount of attention pipelined query processing has received.

Our paper is organized as follows: In Sections II and III we cover essential background and discuss related work. We discuss the dimensions associated with this study in Section IV, and present our analytical model in Section V. We compare memory footprints of strategies with different UoTs in Section VI. In Section VII, we present our experimental evaluations, and Section VIII contains our concluding remarks.

II. Background and Related Work

In this section we describe the basics of data transfer mechanisms for query processing, which we then use to set the discussion for the rest of the paper.

Data-transfer mechanisms: Since most related works in this area use the word 'pipeline', we will first describe a 'pipeline' so that we can refer to previous work using their own terminology. However, we emphasize that a pipeline is simply one of the many possible data transfer mechanisms.

A minimal pipeline in a query plan consists of two operators: A *producer* operator and a *consumer* operator. The output of the producer operator can be passed (or *streamed*) to the consumer operator, even when the work for the producer operator has not finished. An example of a simple pipeline is a query plan with a logical select operator feeding into a logical hash-based join operator. Here, a (physical) select operation (on the probe side) feeds into a "probe" operation.

Deeper pipelines may consist of more than two operators, such that any two adjacent operators can form a producer and a consumer pair. Data from the original producer operator can be passed all the way to the last operator in the pipeline. An example of a deep pipeline is a left-deep plan for a multi-way hash join query plan, with all hash tables on the build side being resident in memory.

There are two key aspects about pipelining, or in general data transfer mechanisms: *Materialization* (or the lack of) and *eager* execution of consumer operator on the output of a producer operator. Different systems may vary in the representation that is used for the temporary data, which is the output of a producer operator. Systems such as MonetDB [23] and Quickstep that employ a block-style query processing model fully materialize the output. Vectorwise [44] has a compact representation of the intermediate output and does not fully materialize the output.

Data-centric systems such as Hyper [25] and LegoBase [28] generate compiled code for the full pipeline. Compared to block-based systems, like Quickstep, they do not need an explicit representation for the temporary data (the code generator picks the internal representation). One can think of the UoT for such data-centric systems as a single tuple.

Prior Work: Pipelining in database systems has been studied extensively. Wang et al. [40] proposed an iterator model for pipelining in an in-memory database cluster. Their key idea is to provide flexibility in the traditional iterator through operations such as expand and shrink. Neumann [34] proposed compilation techniques for query plans, which is used by Hyper [25], [30]. As discussed earlier, query compilation is one of the techniques for realizing pipelining in a query plan. Vectorwise [44] pioneered the vectorized query processing model through the hyper-pipelining query execution [7]. Departing from the traditional tuple-at-a-time processing model, Vectorwise used batches (or vectors) of tuples. These batches, potentially amenable to using SIMD instructions, help improve Vectorwise's performance over its predecessor MonetDB [23].

Kersten et al. in their work on query compilation and vectorization [26] provide a summary of pipelining in many systems, from systems as old as System-R [27] to modern systems like Hekaton [17]. The authors describe two approaches to pipelining, namely the pull (*next* interface) and push (*producer/consumer* interface) model. Quickstep uses the push model of pipelining.

Menon et al. proposed *Relaxed Operator Fusion* model [32] to bring together techniques like compilation, vectorization and software prefetching in a single query processing engine Peloton [2]. Funke et al. showed [18] how pipelined query processing can work with query compilation and GPU accelerated database systems.

There is a large body of prior work on the effect of storage formats and page layouts on query performance [8], [5], [21], [20], [4]. We focus on using row and column store formats for the comparison between various pipelining strategies.

Incorporating parallelism for query execution within a single node database system has been an active area of study since the prevalence of multi-core computing, which is exemplified by many modern systems including [36], [30], [44], [1].

Liu and Rundensteiner [31] studied pipelined parallelism in bushy plans and propose alternatives to aggressively leverage pipeline processing. Their work focuses on optimizing query plans in a distributed execution environment with limited

memory per node. Our work differs from them in multiple aspects: We focus on single node in-memory query execution with large intra-operator parallelism. Further, we focus on the query scheduler phase, which comes after the (optimized) query plan has been generated by the optimizer.

Zhu et al. [42] proposed *look ahead techniques* to increase the robustness of join query plans, allowing for efficient query execution in many cases even when the join order may be "sub-optimal." Their key idea is to reduce the data movement between a producer operator and a consumer operator in a join pipeline by employing a sequence of bloom filters.

Pipelines in many TPC-H queries begin by filtering a large table (e.g., the lineitem table). Researchers have looked at sharing this large amount of work across multiple queries [22], [43]. Scan sharing has shown significant improvements in query performance, especially in the disk setting.

Many commercial systems including SQL Server [29], Oracle [11], IBM DB2 [37], Snowflake [12] make use of pipelining. SQL Server's query progress estimation techniques rely on pipelines within a query plan [29], [9].

Distributed systems (MapReduce [13], Dryad [24]) favor reliability over pipelining; thus, they materialize the intermediate data during a job. Recently *Bubble Execution* [41] proposed breaking a query execution plan in *bubbles* such that data can be streamed within a bubble, while offering reliability.

III. Quickstep Background

In this section we provide a brief description of Quickstep and its implementation of different data transfer mechanism strategies. We introduce the system to facilitate the subsequent discussion on various dimensions and the experimental results.

Quickstep aims for high performance for in-memory analytic workloads on a single node. One of the techniques used by Quickstep to get high performance is large intra-operator parallelism. Quickstep uses a cost-based optimizer to generate query plans. Joins in Quickstep use non-partitioned hash-based implementation [6]. The operators in Quickstep process a batch of input tuples, rather than one tuple at a time. Prior work [7] has shown that the vectorized style processing outperforms tuple-at-a-time processing technique.

Quickstep uses an abstraction called *work orders*, which represents the relational operator logic that needs to be executed on a specified input. The work done for a query is broken into a series of work orders. These work orders can then be executed independently and in parallel.

There are two kinds of threads in Quickstep – *worker* and *scheduler*. Worker threads execute work orders. Once assigned a work order, the worker thread executes it until its completion. A single scheduler thread coordinates the execution of work orders, including work dispatch and progress monitoring.

A. Managing Storage in Quickstep

Quickstep supports a variety of storage formats such as row and column store with an optional support for compression. The data in a table is horizontally partitioned in small independent storage blocks. The size of each storage block is fixed, yet configurable. The intermediate output of relational operators (e.g. filter) is stored in temporary output blocks, which follow a similar design as the storage blocks of the base tables.

Each relational operator work order has a unique set of input, described based on the semantics of the operator. For instance, a select work order's input consists of a storage block and a filter predicate. A probe join hash table work order's input is made up of a pointer to the hash table and a probe input block. A work order execution involves reading the input(s), applying the relational operator logic on the input(s), and finally writing the output to a temporary block. (The output of most operators is represented in the form of storage block, except when the output itself is a data structure like hash table; e.g., in the case of a build hash operator, or hash-based aggregation operators.)

Quickstep maintains a thread-safe global pool of partially filled temporary storage blocks. During a work order execution, a worker thread *checks out* a block from the pool, writes the output of the work order to the block, and returns the block to the pool at the end of the work order execution. Thus, a block is used by at most one operator work order at any given point in time. This approach has two benefits: 1) We maintain locality of the output block when output is written to it, and 2) Reduced memory fragmentation due to the reuse of output memory blocks.

B. Unit of Transfer (UoT)

As Quickstep is fundamentally built on a block-based storage architecture, the UoT used in Quickstep is also defined w.r.t. blocks. As described earlier, the output of a relational operator work order is stored in temporary blocks. As soon as a block is full, it may be deemed ready for data transfer, subject to the UoT value. For a small UoT value, the scheduler receives a signal as soon as an intermediate output block is full, after which it dispatches a work order for the consumer operator for execution. Partially filled blocks are scheduled for data transfer at the end of the operator's execution.

Interplay between block size and UoT: For a given block size, we consider two extreme values for UoT. The smallest UoT is a single block. As soon as a block is produced, we transfer it to the consumer. The largest UoT is the entire intermediate table, and in this case the system waits for the entire table to be produced before making it available to the consumer.

C. Data Transfer Mechanism and Scheduling

A scheduling strategy in Quickstep determines the sequence in which work orders of a query are executed. As shown in Figure 2, different values of UoTs generate different scheduling strategies.

For smaller UoT values, a consumer operator work order is scheduled as soon as it is available. At the higher end of the spectrum of UoT values, a consumer operator work order is not scheduled until all the corresponding producer work orders have finished execution.

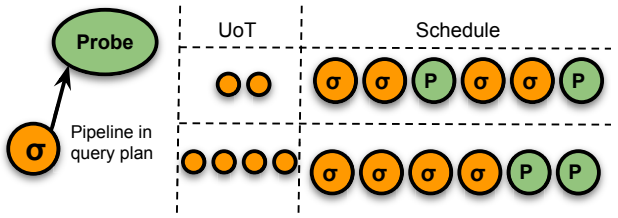


Fig. 2: **Interplay between scheduling strategies and UoT values. A filter operator (σ) and a probe operator (P) for a hash join are shown on the left. We compare two scenarios: A low UoT value (with two blocks in a UoT) and a high UoT value (with 4 blocks), while keeping the block size as same. On the right are two schedules of work orders. As the number of blocks in the UoT increases, the schedule starts to look like traditional non-pipelining approach.**

The Quickstep scheduler allows development of sophisticated scheduling policies [14], such as implementation of an operator with an upper or lower limit on the number of concurrent consumer work orders under execution, or executing operators under a specified memory budget.

IV. Discussion on Dimensions

In this section we identify the dimensions that may have an impact on the performance of data transfer mechanisms for different values of UoTs. We classify these dimensions into three categories: *physical organization of data* (storage format and block size), *execution environment* (parallelism and hardware characteristics), and *structural aspects of query*. We describe these dimensions below.

A. Block Size

We first explain the concept of a block size. As the producer operator processes the input, it materializes the output to a temporary block. The block size in Quickstep for a given table is fixed, and can be specified at the time of its creation.

We are interested in the impact of block size on the performance of the data transfer mechanisms. Consider data transfer between two operators: `select` operator $\rightarrow$ `probe` hash table operator. A smaller output block size means that the block can potentially fill quickly, resulting in more `probe` work orders.

B. Storage Format

Data processing time is impacted by the way data is organized. We look at two common storage formats: the row store format and the column store format. In the column store format, values for a given column are stored in a contiguous memory region. Scanning a single column results in a sequential memory access pattern, and generally good cache behavior. In the row store format, all the columns of a tuple are stored in a contiguous region. Thus, scanning a particular column involves bringing unnecessary data (non-referenced columns) into the caches. Selecting all the columns in a row, however, is more efficient.

Prior work has shown that column stores deliver better query performance for analytical workloads [4], especially for scan operators. Recent studies [36] have shown that the performance gap between column stores and row stores is not as high as shown in previous work. Therefore we explore both storage formats. For our comparison, we assume that all base tables are stored in the same storage format. For micro-benchmarking in Quickstep, we note that the row store format is used for temporary tables irrespective of the storage format of the base tables.

C. Parallelism

We focus on two kinds of parallelisms in query processing: a) Inter-operator: processing multiple operators at the same time [23], [30], [36], [44], [11], [37], and b) Intra-operator: parallel processing inside a single operator. Note that these two kinds of parallelisms can co-exist in a system [44], [30], [36]. In this paper, we study the impact of intra-operator parallelism on the relative performance of data transfer mechanisms with different UoT values.

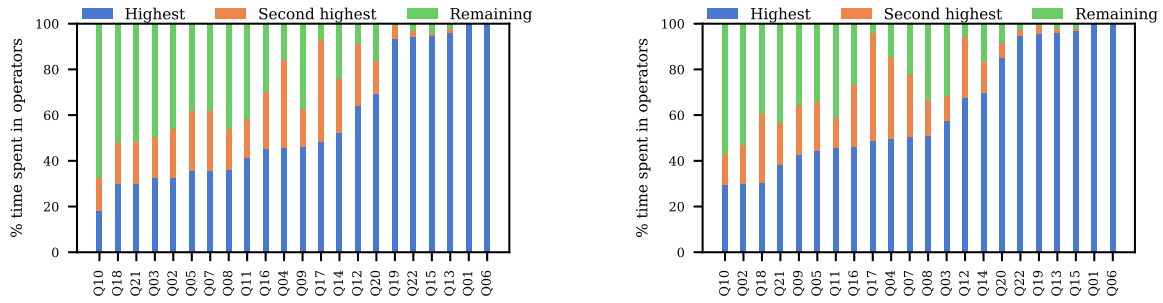
1) Degree of parallelism: The degree of parallelism (DOP) of an operator refers to the number of concurrent threads involved in executing work orders of that operator. The *scalability* of an operator (using T threads) is its performance with DOP as T relative to its performance when DOP is 1.

2) Intra-operator parallelism in Quickstep: The scheduler of Quickstep, as discussed before, dispatches work orders of relational operators to worker threads. Thus, the DOP of an operator at a given instance is the number of its work orders under execution. Since the number of work orders of an operator can change over time, the corresponding DOP also changes over time dynamically.

3) Interplay between DOP and UoT values: The UoT value used in query processing can have an impact on the DOP of the operators. Consider the example from Figure 2 which compares two UoT values while keeping the block size constant. In the small UoT case, the available CPU resources are shared among the filter and probe work orders. For the higher UoT value, the CPU resources are used exclusively; first by the filter work orders first and then by the probe work orders. Thus, the DOP of the operators is lower for small values of UoT.

4) Scalability: In theory, adding more CPU resources for an operator execution should offer linear speedup. The assumption being that each parallel work order operates at the same speed and thus by executing more work orders concurrently, the overall execution time reduces proportionally.

Linear speedups for operators (or for queries as a whole) are not always possible. DeWitt and Gray [16] propose reasons for less than ideal speedup for parallel databases such as startup costs, interference from concurrent execution, and skew. We can extend some of their ideas to in-memory systems. For example interference can come from various sources such as contention due to latches, and shared use of a common bandwidth in a memory bus, or shared channels for data movement across NUMA sockets.



(a) **Column store**

(b) **Row store**

Fig. 3: **Distribution of the time spent in each TPC-H (scale factor 50) query in operators.**

For an operator that exhibits poor scalability, increasing its DOP beyond a certain limit may degrade its performance. Specifically, the execution time for each work order of the operator may increase with a higher DOP value. Recall from Section IV-C3, the DOP of a consumer operator is proportional to the UoT value. Thus, smaller values of UoT are good for the performance of operators with poor scalability.

D. Hardware Prefetching

Hardware prefetching is a technique used by modern processors to proactively fetch data in caches by speculating its access in the future. The prefetcher observes patterns of data accesses from memory to caches and speculates the access of a data element in advance. Prefetching hides the latency due to a cache miss and potentially improves performance. There are two kinds of prefetching: spatial and temporal, and in this paper, we focus on spatial prefetching.

Next we describe why prefetching is important in our study. Lower values of UoT generally results in a large number of context switches for work order execution (cf. Figure 2). Thus, having a lower value of UoT may affect the hardware prefetcher's ability to predict the data access patterns. Therefore, we are specifically interested in the impact of hardware prefetching at lower UoT values.

In addition to the hardware-based prefetching implementation there are software-based techniques for prefetching, which can be used to improve the performance of relational operators [10], [32]. By focusing on hardware-based prefetching, we can observe the impact of the hardware prefetcher without modifying the implementation of the relational operators.

For our study, we run the queries with pipelining in two scenarios: a) when hardware prefetching is enabled (this is the default behavior of the hardware), b) when hardware prefetching is disabled (by setting bit 0 and 1 in Model-specific Register (MSR) at address 0x1A4) as per Intel's guidelines [39].

E. Query Plan Structure

Complex queries like the ones in TPC-H contain several operators, and the impact of UoT values on overall query execution time is not immediately evident.

To analyze the impact of UoT values on the response time of a query, we conduct an experiment to dissect the time distribution of the execution of TPC-H queries. We focus on the most dominant operator (where the most of the execution time is spent) and the second most dominant operator for each query. Note that for this analysis, we run the queries with a high UoT value (the whole table) to avoid any overlap in time. The intuition is that if there is only one operator in the query where the majority of the query execution time is spent, small UoT values may not play a big role in the overall execution time of the query.

Figure 3a shows the results of this experiment for base tables stored in a column store format. For some queries (Q1, Q6, Q13, Q14, Q15, Q19, Q22) the dominant operator takes up the majority of the query execution time (more than 50%). We also note that the dominant operator for many of these queries is a "leaf" operator (e.g. selection on a base table, building a hash table on a base table, aggregation on a base table). Therefore, depending on the structure of a specific query, small UoT values may not provide significant advantage in improving the query execution time.

Further, at times large data is pruned at the initial operators e.g, due to a highly selective filter predicate or a join condition, or due to application of sideways filters (e.g. LIP [42]). In such cases, very little data is passed to the consumer operators, and consequently, the impact of low UoT values is not significant.

V. Analytical Model

In this section we analytically model the performance difference for varying UoT values. The model uses the dimensions introduced in the previous section including the number of threads used for execution (parallelism), the UoT values, memory/cache access times, and cache miss penalties (hardware prefetching). Our model is targeted towards in-memory environments, but it can be extended to other storage device settings, as we show in Section V-C. The model and analysis of memory usage differences is presented in the next section.

Notation	Description
R_h	Cost of reading an UoT to memory hierarchy h from a lower hierarchy h + 1
AR_h	Amortized cost of reading an UoT sequentially to memory hierarchy h from a lower hierarchy h + 1
W_h	Cost of writing an UoT to memory hierarchy h from a higher level hierarchy
IC	Cost of an instruction cache miss
M_h	Cost of missing a UoT at memory hierarchy h
N_{op}^{in}	Number of input UoTs for operator op
N_{op}^{out}	Number of output UoTs for operator op
T	Number of threads in the system
B	UoT size

TABLE I: **Notations used for the analytical model**

The key idea that we employ is to focus on operations that result in a cost difference and to ignore common operations that occur irrespective of the UoT values. Many operations are common to query processing for all UoT values: e.g., the total cost of reading from an L1 cache line is the same irrespective of the UoT value. As we are interested in the relative comparison of performance between two different values of UoT, we largely focus on the additional work that is incurred when using one UoT value over another UoT value.

Additionally, we take into account the benefits of hardware prefetching when reading multi-megabyte blocks as UoT in a sequential access pattern; the amortized cost of reading a UoT will be substantial smaller than the cost when each UoT is read on its own without prefetching. As the UoT is read into memory, access to the initial tuples likely incur an L3 cache miss, but we assume that the prefetcher can quickly detect the access pattern, and thus the miss penalty will decrease quickly.

We analyze a basic producer-consumer pair, in which the producer is a select operator and the consumer is a probe operator for a hash-based join operation. This producer-consumer pair is commonly found in the query plans of analytic workloads, such as TPC-H queries. For example, in the query plans for Q07 and Q19 from the TPC-H benchmark, selection is performed on the *lineitem* table and the output is subsequently used to probe a join hash table, forming the `select`→`probe` pair for data transfer. Table I contains various parameters that we use to determine the costs for different scheduling strategies.

For high UoT values which are comparable to the size of the table, the output of the `select` operator is not immediately consumed by the `probe` operator. The `probe` operator is only initiated after the `select` operator is complete. Thus, writing the output of the `select` operator to memory, and reading the same UoTs as input to the `probe` operator is additional work in the non-pipelining case. Moreover, an input probe UoT is likely to be cold in the caches when it is read for the `probe`.

Thus, for the case of high UoT values equal to the size of the table, the extra work done can be quantified as:

$$W_{mem} \cdot N_{select}^{out} + AR_{L3} \cdot N_{probe}^{in} + p_1 \cdot N_{probe}^{in} \cdot M_{L3}$$

Here, $W_{mem} \cdot N_{select}^{out}$ is the cost to write the output of the `select` operator from cache to memory.

Further, $AR_{L3} \cdot N_{probe}^{in}$ is the total cost of reading `probe` UoTs sequentially from memory, expressed as the amortized cost of reading a UoT sequentially many times.

For the last term, note that a `probe` work order has two input components: `probe` input UoT and a hash table. As the reads to a hash table are random, it disrupts the sequential access pattern used to read the `probe` input UoTs. Therefore, we account for the cost to read the `probe` input UoTs as $p_1 \cdot N_{probe}^{in} \cdot M_{L3}$, where p_1 is the probability that there is a L3 cache miss when reading the `probe` input after the context switch back from reading the hash table.

Next, we quantify the additional work done for the case of small UoT values. In this case, the input for `probe` UoTs (which is the output of `select`) is presumed to be resident in processor caches. This assumption leads to the following model:

$$(N_{select}^{out} + N_{probe}^{in}) \cdot IC + p_2 \cdot N_{probe}^{in} \cdot (M_{L3} + R_{L3}) + p_1' \cdot (M_{L3} + R_{L3} + W_{mem}) \cdot N_{probe}^{in}$$

Notice that for low UoT values, every `probe` work order execution involves two context switches: First from the `select` operation to the `probe` operation and another from the `probe` operation to the `select` operation. Thus we account for two instruction cache misses; one for each context switch, which is represented by the term: $(N_{select}^{out} + N_{probe}^{in}) \cdot IC$.

Now, we explain the term $p_2 \cdot N_{probe}^{in} \cdot M_{L3}$. It represents the cache misses due to the disruption in sequential access pattern of the `select` operation, and is caused by the intermittent `probe` operations. The term p_2 is the probability of an L3 cache miss for the `select` operator after the context switches back from the `probe` operation. The term $p_2 \cdot N_{probe}^{in} \cdot R_{L3}$ represents the time taken to move data from memory to cache after encountering a cache miss.

Finally, the term $p_1' \cdot (M_{L3} + R_{L3} + W_{mem}) \cdot N_{probe}^{in}$ is analogous to the L3 cache misses incurred when the probe input block is read in the non-pipelining case. Here the assumption that the probe inputs are resident in the L3 cache. Thus, the probability of whether a probe input is read "hot" or not is dependent on the size of UoT.

Due to factors such as reading in the relevant UoTs of the hash table for a probe operation and multiple threads sharing the L3 cache, each write operation that is incurred when creating a `probe` input, and the subsequent `probe` input read operation is not guaranteed to be served from the L3 cache; this cost is exacerbated with larger values of UoT. To account for this cost, the term p_1' is used to represent the likelihood that the reads and writes incur L3 cache misses; it is expressed as $min(1, 2B \cdot T/size(L3))$. The term p_1' is smaller for small UoTs, and it is 1 for high values of UoTs and when T is high.

A. Quantifying the Difference

We now quantify the differences between the two extreme values of UoTs. We first make a few observations to simplify the analysis. As large UoT values are typically a few megabytes, the instruction cache miss costs become negligible

in this case. Thus, we can ignore the cost associated with instruction cache misses. Second, we observe that $N^{in}_{probe} = N^{out}_{select}$. Thus, the ratio of costs of non-pipelining (informally large UoT) and pipelining (informally low UoT) strategies is as follows:

$$\frac{W_{mem}\cdot N^{in}_{probe}+AR_{L3}\cdot N^{in}_{probe}+p_1\cdot N^{in}_{probe}\cdot M_{L3}}{p_2\cdot N^{in}_{probe}\cdot(M_{L3}+R_{L3})+p'_1\cdot(M_{L3}+R_{L3}+W_{mem})\cdot N^{in}_{probe}}$$

This ratio can be simplified to:

$$\frac{AR_{L3}+W_{mem}+p_1\cdot M_{L3}}{p_2\cdot(M_{L3}+R_{L3})+p'_1\cdot(M_{L3}+R_{L3}+W_{mem})} \quad (1)$$

Observe that $AR_{L3} \ll R_{L3}$, while both costs, AR_{L3} and R_{L3}, are directly proportional to the size of UoT, B. We consider the two representative cases of high and low UoTs values to estimate the difference between the two strategies.

*a) **High UoT values**:* For high UoT values (size $> \frac{|L_3|}{2\cdot T}$), p'_1 is close to 1, and p_2 will have a low value. Additionally, the cost contribution of M_{L3} is low in general, and W_{mem} becomes the dominant cost. We expect that $p_1\cdot M_{L_3} \approx M_{L3}\cdot(p'_1+p_2)$; $p_2\cdot R_{L3}+p'_1\cdot(R_{L3}+W_{mem}) \approx p'_1\cdot(R_{L3}+W_{mem})$, which leads to $p'_1\cdot(R_{L3}+W_{mem}) \approx AR_{L3}+W_{mem}$. Hence, the ratio given in Equation 1 will be very close to 1. Thus, we expect for high UoT values, the difference between the two strategies to be negligible.

*b) **Low UoT values**:* Smaller UoT values result in a large number of work orders, which can incur a large overhead in storage management. Some examples of such overhead include creation cost of several UoTs, maintaining references for UoTs present in-memory, and synchronisation costs in the data structures for storage management. So, in this scenario, p_2 will be close to 1, and p'_1 will have a lower value, though not negligible. The cost contributions from the terms AR_{L3}, $p_2\cdot M_{L3}$, $p_1\cdot M_{L3}$, and $p'_1\cdot M_{L3}$ will not be significant, and we expect that $W_{mem} \approx p_2\cdot R_{L3}+p'_1\cdot(R_{L3}+W_{mem})$. The ratio will be very close to 1; since the cost of W_{mem} is dominant, the cost of $p_2\cdot R_{L3}+p'_1\cdot(R_{L3}+W_{mem})$ can be slightly lower than W_{mem}, giving the execution with lower UoT values a slight advantage.

B. Generalization to other pipelines

So far we have considered only the `select` → `probe` operator pipeline. We focused on this particular pipeline because it is found in many analytic queries. Some other operators in a query plan are `sort`, `sort-merge join`, `sort-based aggregation`, `hash-based aggregation` and `nested loops join`[1]. Sort-based operations are typically blocking and generally not amenable to pipelining. Hash-based aggregation is similar to the hash-probe scenario that we have discussed. For `nested loops join`, the UoT values determine how often there are cache misses due to context switches for the outer relation. For the inner relation, nested loops join involves sequential access pattern. In these other pipelines, we hypothesize that the performance for high UoT values and low UoT values will be similar, as the cost of cache misses resulting from context switches would be offset by the other access pattern that is sequential.

[1]We could not experimentally validate our hypothesis as in TPC-H queries, Quickstep optimizer does not produce plans with such operator pipelines

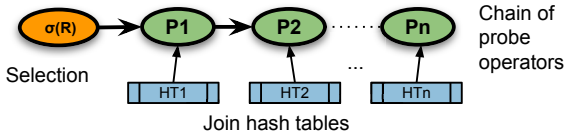


Fig. 4: **A left-deep query plan fragment showing a cascade of selection and multiple probe operators**

C. Applying Model to Other Storage Settings

Our model can be easily applied to other settings, such as storing data in a persistent store (such as SSD and hard disk drives) with a in-memory buffer pool. We change the parameters from Table I appropriately to fit the persistent store setting. The terms p_1 and p_2 are nearly 0, when the hash table is nearly always kept in the buffer pool. Thus, the additional cost incurred for large values of UoT is:

$$R_{store}\cdot N^{in}_{probe}+w_{store}\cdot N^{out}_{select}$$

which could be in the order of seconds for thousands of UoTs. The additional cost incurred for lower UoT values is:

$$N^{out}_{select}\cdot IC+N^{in}_{probe}\cdot IC$$

Note that this value is substantially lower (order of nanoseconds or microseconds for thousands of blocks) than that in the non-pipelining case. Thus, the analytical model is consistent with the expected behavior for persistent store-based systems.

VI. Memory Characteristics

We now discuss the memory footprint for different UoT values. We first formulate the memory footprint of two extreme UoT values – a high value which is equal the size of the input table, and a low value. The choice of our UoT values mimic what are traditionally known as "blocking" and "pipelining" modes of query processing. Subsequently, we compare their memory behavior against each other.

As an example, let us consider a cascade of selection and multiple probe operators as shown in Figure 4. For low UoT values, once we read a tuple, it is processed by the selection operation first, and if selected, it is further processed by the subsequent probe operators (subject to the join condition). This pipeline necessitates the construction of all hash tables before the execution of selection-probe operators.

We now consider a large UoT value which is equal to the size of the selection operator's output. For such a large UoT value, the execution can be described as "one join at a time". The selection operation is completed first, followed by building of the hash table and then the probe. Thus, only one hash table needs to be created at any point of time. However this case of execution materializes the result of the selection (and successive probe operations).

Strategy	Memory footprint	
	Hash table	Intermediate table
Low UoT value	$\sum_{i=1}^{n} \|H_i\|$	0
High UoT value	$\|H_1\|$	$\|\sigma(R)\|$

TABLE II: **Memory footprint for low and high UoT values**

Query	Selectivity (%)	Projectivity (%)	Total (%)
03	53.9	13.1	7.0
07	30.4	18.3	5.6
10	24.7	13.1	3.2
19	2.1	13.1	0.3
Average	27.8	14.4	4.0

TABLE III: **Memory reduction with input table lineitem**

We contrast the memory requirement for the leaf level join tree in Table II. We denote the size of the i^{th} join hash table by $|H_i|$. The size of the selection output is denoted by $|\sigma(R)|$ where R is the input table.

We disregard the common elements contributing to the memory footprint to determine the difference of memory footprints such as current join hash tables, base tables, final join output. Note that our analysis is done on the leaf level join, however it can be extended to any intermediate join easily. Therefore the memory overhead comparison for the two strategies is as follows. For low UoT values: $\sum_{i=2}^{n} |H_i|$, and for high UoT values it is $\sigma(R)$.

A. Memory Overhead for high UoT values

We now dig deeper into the memory overhead caused by having a large UoT value. The key relationship is between the size of the base table and the size of the materialized intermediate table. Typically a selection operation on a base table causes reduction in memory in two ways. The first and the obvious aspect is the selectivity of the filter predicate. We define selectivity as $s = N_s/N$, where N_s is the number of rows that pass the filter predicate and N is the number of rows in the input table. The other aspect is projectivity, which we define as $p = C_s/C$, where C_s is the total size of the columns projected per tuple and C is the total size of the columns in the base table per tuple. We compute selectivity and projectivity relative to the size of the input table.

B. Memory Overhead for low UoT values

As described earlier, the memory overhead for low UoT values is the combined memory of the hash tables that can be probed (except the current join).

Let us consider a single hash table. Typically hash tables have fixed-sized buckets, say c bytes. Hash tables also have a load factor, say f where $f \in (0, 1]$. If f is 0.5, the hash table gets resized as soon as its memory occupancy reaches 50%. Therefore, the memory footprint of each entry that is inserted in the hash table is c/f. If the input tuple has a size of w bytes, and the input table's size is M bytes, then the resulting hash table size is $(M/w) \cdot (c/f)$. For low UoT values, computing the memory overhead involves summing the hash table sizes for all the hash tables involved.

Query	Selectivity (%)	Projectivity (%)	Total (%)
03	48.6	8.7	4.2
04	3.8	10.9	0.4
05	15.2	5.8	0.9
08	30.4	11.6	3.5
10	3.8	5.8	0.2
21	48.7	2.9	1.4
Average	25.1	7.6	1.8

TABLE IV: **Memory reduction with input table orders**

Now that we have established the memory overhead for both low and high UoT values, a natural follow up question is: Which UoT values result in lower memory overheads? The answer is dependent on the query and its input tables characteristics. On the one hand, we see many cases when a lower UoT value results in a lower memory footprint compared to using a higher UoT value, especially for queries in the Star Schema Benchmark (SSB) [35] that have small join hash tables. On the other hand, we show in the next section that sometimes high UoT values can also result in significantly low memory overheads.

C. Memory Analysis for TPC-H Queries

We report the selectivity and projectivity values for selected TPC-H queries that contain a selection and probe pipeline of operators when the base table is `lineitem` or `orders` (the two largest tables in the TPC-H schema).

Table III and Table IV presents the selectivity, projectivity and overall memory footprint of selection operations in selected TPC-H queries. A key takeaway from these tables is that even though the selectivity is high, due to the projectivity, the relative memory overhead of a select operation can be quite low. In a star-schema or a snowflake-schema typically the fact tables have large number of rows as well as a large number of columns. If few columns are projected from the fact table during a selection operation, then the relative memory overhead can be low. Note that both selectivity and projectivity numbers reported above are without any optimization, thus they are on the higher side. In practice there are many techniques to reduce both selectivity and projectivity, as discussed next.

Techniques to lower selectivity: Aggressive pruning techniques like Lookahead Information Passing (LIP) filters [42], can substantially bring down the selectivity, sometimes by an order of magnitude or more. Query optimizers often push down predicates, which can also reduce the selectivity.

Techniques to lower projectivity: One way to lower the projectivity is to trade memory for computation. Consider the expression `l_extendedprice * (1 - l_discount)`, which is found in many TPC-H queries. A lazy evaluation of this expression involves projecting `l_extendedprice` and `l_discount` attributes. However if the evaluation is folded with the selection operation, we can project only one attribute, which is the resultant expression.

To compare the memory overhead between low and high UoT values, consider TPC-H Q07. This query has a selection operation on `lineitem` followed by a cascade of three probe operations. Of the three hash tables required in this data

Parameter	Description
Processor	2 Intel Xeon Intel E5-2660 v3 2.60 GHz (Haswell EP)
Cores	Per socket – 10 physical, 20 hyper-threading contexts
Memory	80 GB per NUMA socket, 160 GB total
Caches	L3: 25 MB, L2: 256 KB, L1 (both instruction and data): 32 KB
OS	Ubuntu 14.04.1 LTS
Data set	TPC-H [3] data (and queries) for scale factor 50
Block sizes	128 KB, which is half of the per-core private L2 cache size, and 2 MB, which comfortably fits in L3 cache.
UoT values	Low UoT is the same as block size and high UoT is the same as full table's size.

TABLE V: **Evaluation platform**

transfer cascade, the second hash table is built on the entire `orders` table, which is around 2.4 GB in Quickstep for a scale factor of 100. The intermediate output of the selection operation is 2.8 GB without any optimization and 224 MB with bloom filter based pruning [42]. Thus, in some cases the memory overhead caused by lower UoT values can be substantially higher compared to the memory overhead caused by higher UoT values.

VII. Experimental Evaluation

We now apply the proposal in Section V and study the query performance implication of UoT values. Our goal is to understand the performance characteristics of queries for different UoT values, and while doing so, to observe the impact of the various dimensions discussed in Section IV.

A. Experimental Setup

Table V summarize our experimental setup. Note that we use only a single NUMA socket in this study. We also experimented with block size as 512 KB and observed similar results as reported below. Unless specified, we report the results for the column store format and 20 threads as Quickstep workers. The buffer pool size in Quickstep is configured with 80% of the system's memory (126 GB). We run each query 10 times and report the mean of the best three runs.

B. Results

First, we analyze the performance of singleton operators, followed by a more complex pipeline of operators. Finally, we study the execution time of entire queries.

1) Performance of Consumer Operators: Low UoT values ensure that the consumer operator's input is "hot" in caches. Does the hotness of the input improve the performance of the consumer operator? To investigate this issue we perform the following experiment.

We focus on key deep operator chains (select→probe, as shown in Figure 4) from the TPC-H queries.

Figure 5 plots the work order execution times for the first consumer operator. We can observe that a low UoT value generally benefits the performance of the probe operator. The extent of improvement diminishes as we increase the block size from 128 KB to 2 MB. This behavior is consistent with the findings from the analytical model (cf. Section V).

2) Performance of Operator Chains: Having looked at the performance of the consumer operators, we zoom out to look at the performance of the complete chain of operators.

Figure 6 shows the results for this experiment. For smaller block sizes, low UoT outperforms high UoT values only in some queries. These queries are the ones in which a low UoT value has a superior operator performance (see Figure 5). At a 2 MB block size, the operator chains from all queries perform equally well with both UoT values.

Notice that despite low UoT values having an edge for individual consumer operator performance, the extent of improvement does not match in the operator chain performance. This behavior is because typically a chain has other operators (producers, which often dominate the overall execution time).

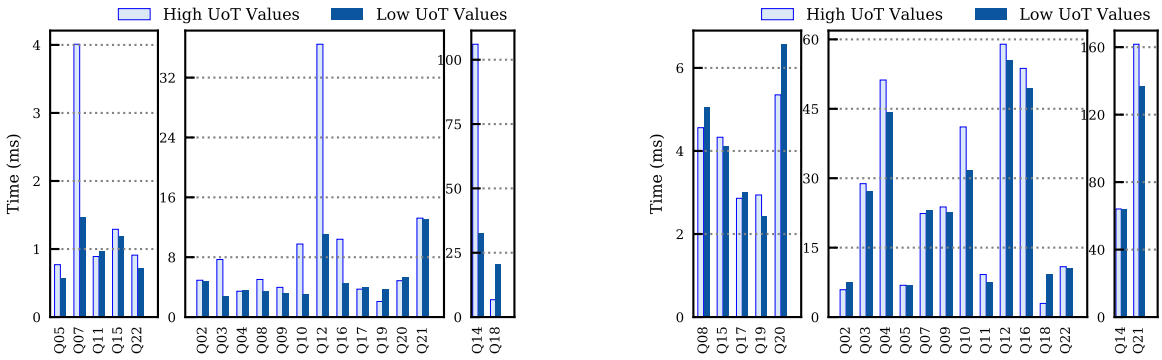
3) Overall Execution Time:: After analyzing the performance of deep operator chains, we further zoom out to the execution times of the full queries. Figure 7 shows the results for query execution times for different UoT values. We can observe that low UoT values perform slightly better for smaller block sizes. As the block size increases, there is little difference between the two alternatives. From these experiments we can conclude that although having low UoT values benefit individual operators, their impact on overall query performance is insignificant.

An alert reader may have noticed that query performance improves as the block size increases for both UoT values. This behavior is due to Quickstep's design which favors large multi-megabyte blocks. For smaller blocks, the storage management and work order scheduling overheads increase. However, that this is an orthogonal issue and doesn't affect our overall observations, which is also supported by the fact that the performance impact of block size variation is similar for both ends of the spectrum of UoT values.

4) Effect of Storage Format: Next, we study the effect of storage format of base tables on the performance of using low UoT values. We use two configurations, each with block size of 2 MB, and all tables either stored in a) column store format or b) row store format. Note that in both configurations, the temporary tables are stored in a row store format.

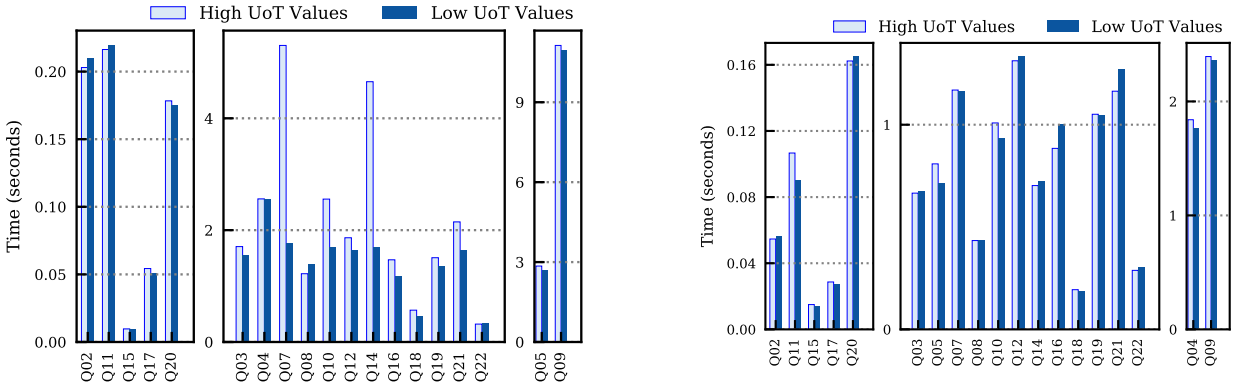
Performance comparison between pipelining strategies with row store: Recall the trends we discussed for the performances of consumer operator (Section VII-B1), operator chain (Section VII-B2) and entire query (Section VII-B3) for column store. We observe similar trends for the row store case. In the interest of space we present only one graph for query execution time using 2 MB block size in Figure 8.

When we compare Figure 8 (row store) with Figure 7b, we make two observations. First, the query performance is unaffected by the choice of the UoT value. Second, queries perform better with tables stored in column store. One reason why scans on row stores are slower than scans on column store is that processing a tuple involves fetching unnecessary data. In contrast, with column stores, we only process what is necessary from an attribute, and avoid fetching unnecessary data to the caches.



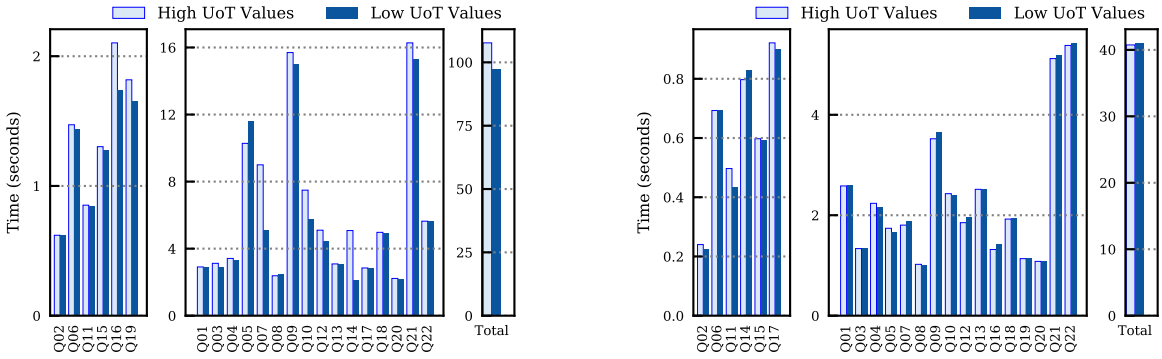
(a) Block size 128 KB (b) Block size 2 MB

Fig. 5: **Per-task execution times of the probe hash operator when it is the first consumer operator in a pipeline**



(a) Block size 128 KB (b) Block size 2 MB

Fig. 6: **Execution times of operator pipelines**



(a) Block size 128 KB (b) Block size 2 MB

Fig. 7: **Execution times of queries.**

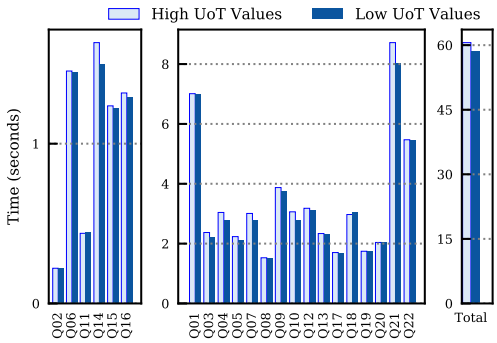


Fig. 8: **Execution times of TPC-H queries for the row store format and a block size of 2 MB**

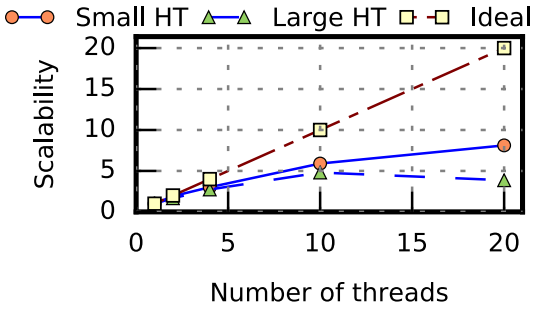


Fig. 9: **Scalability of two probe operators with different hash table sizes from TPC-H Q7. Ideal scalability included.**

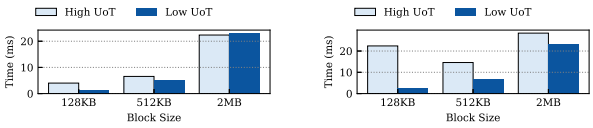
5) Effect of Parallelism: Next, we study how intra-operator parallelism impacts performance with the two UoT values that we have considered. In this experiment we also report results from running Quickstep with a block size of 512 KB.

We present results for TPC-H Q07 that shows the impact of scalability on the performance of operators. We pick two probe operators from query Q07, which are part of a single operator chain. One has poor scalability, and the other has slightly better scalability. The reasons for the poor scalability of the probe operator are: a) It probes a large hash table b) The large hash table also brings contention issues in the storage management subsystem. We present the scalability of these probe operators in Figure 9.

Next, we want to know how the choice of a UoT value behaves with these operators? Figure 10a and Figure 10b present the performance of these two operators.

First, let us analyze the operator with the better scalability (whose input hash table is small). As the block size increases, both UoT alternatives keep up with the larger probe load, and the per task execution time increases as expected.

Now we discuss to the operator with poor scalability. Note that as the block size increases from 128 KB to 512 KB, the probe operator's performance improves. As the block size increases, the contention on the storage management reduces,



(a) Performance (per-task execution time) of the probe operator with better scalability

(b) Performance (per-task execution time) of the probe operator with poor scalability

Fig. 10: **Effect of interaction of various dimensions on scalability.**

Block size	Select		Build		Probe	
	Yes	No	Yes	No	Yes	No
128KB	0.06	0.08	2.0	1.9	0.8	0.8
512KB	0.2	0.3	8.5	7.6	2.2	0.9
2MB	1.1	1.5	38.0	32.7	3.9	3.1

TABLE VI: **Average task execution times in millisecond for the prefetching experiment. Yes (No) imply that the hardware prefetcher was enabled (disabled).**

and performance improves. As the block size increases from 512KB to 2 MB, contention is no longer a dominant factor, and the task execution time increases because of the added work in a larger block.

Why don't we see similar behavior for low UoT values? It is because by design, the degree of parallelism for low UoT values is smaller (as explained in Section IV-C3, and thus it is less prone to the contention discussed earlier.

Overall, when using low UoT values, the system is more immune to scalability issues as compared to the high UoT value alternative. Systems can have scalability issues due to various external factors such as hardware interference, slow network and internal factors such as skew, poor implementation of operators. (We also note that as future work it would be interesting to explore if the probe operator's scalability changes when using different hash table structures.)

6) Effect of Hardware Prefetching: Next, we examine the effect of prefetching when using a low UoT value. For this experiment the tables are stored in row store format. We run TPC-H queries with and without the hardware prefetcher. We note that the total workload execution time is only slightly (less than 10%) better when the prefetcher is enabled.

In the row store format, to scan even a single attribute, large amounts of unnecessary data is read. As row store tuples are fixed width[2], the hardware prefetcher can detect the access pattern of scanning a single attribute.

To understand the impact of hardware prefetching on individual operators, we pick three operations from Q07 and compare their execution times with and without prefetching: selection, building hash table for a join operation, and probing a join hash table. Our results are presented in Table VI.

[2]Variable length attributes are stored in a separate region, with a pointer to the region stored in the tuple.

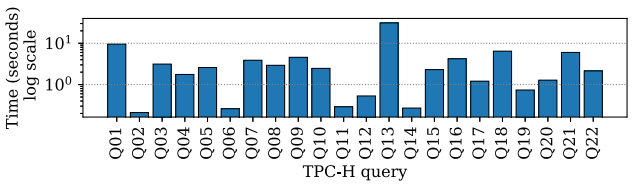


Fig. 11: **TPC-H performance for MonetDB**

Our observations are as follows: Prefetching generally benefits the selection operator. This behavior is understandable as selection has a sequential access pattern and the prefetcher can recognize the strides in the memory accesses across tuples in the row store format.

Prefetching worsens the performance of both the probe and the build hash operator in many settings. For both operators, there are two data streams – a sequential access pattern to read the input and a random read (probe) or random write (build) access pattern on the hash table. We suspect that due to a mix of these two access patterns, prefetching does not benefit these operators.

We ran the same experiment using a column store format, and found little to no performance difference due to prefetching. We speculate that hardware prefetching does not make any significant contribution to an already optimized access pattern of column stores.

7) Experience with MonetDB: MonetDB [23] and Quickstep share some similarities in terms of using a block-style query processing strategy, and materializing the internal state of operators. We evaluate MonetDB's (version 11.41.11) performance on TPC-H queries and show these results in Figure 11.

Comparing the performance of MonetDB with Quickstep (Figure 7b), we can see that in 15 queries, Quickstep performs better than MonetDB. We acknowledge that such a comparison comes with caveats. Quickstep's query scheduler allows easier implementation of various UoT values and scheduling techniques. We are not aware of such a provision in MonetDB. Additionally, Quickstep and MonetDB use different query optimization techniques. For instance LIP filters [42] in Quickstep reduce the data movement across operators significantly.

C. Summary of Experiments

In this section, we summarize our experimental findings and connect them to the dimensions described in Section IV. Recall that a key focus is to understand the relative performance of the two UoT value alternatives at the extreme ends of the spectrum. Our high level conclusion is that in the in-memory setting, for systems using a block-based architecture, the performance is similar for these two alternatives. We now discuss the impact of individual dimensions.

Block size: We find that a larger block size bridges the gap between the performance of the low UoT values and high UoT values. A larger block size results in a lower degree of parallelism for operators in a pipeline, and thus also aides those operators that suffer from poor scalability. However, very large blocks may cause memory fragmentation. It may also result in a low DOP, which could lead to CPU underutilization.

Parallelism: Parallelism can affect the performance of the two UoT alternatives. We saw in Section VII-B5 that using a low UoT value can be more resilient to performance issues arising from poor scalability.

Storage Format: The performance gap between the two UoT alternatives is largely unaffected by the choice of the base table storage format, though some queries execute faster when run on tables in the column store format. The benefit of using column store format over row store format is the highest for base tables (typically leaf operators in a query plan). As the number of attributes in the tables reduce from the leaf operators in a query plans to the root, the advantage of using a column store starts to diminish.

Hardware Prefetching: Hardware prefetching improves performance when using a low UoT value. This effect is more prominent for data stored in a row store format than in a column store format. We saw that prefetching improves scan performance in a representative query. Prefetching had an adverse effect on the probe and build hash operators.

As increasing L3 cache size becomes prohibitive due to power constraints, hardware prefetching techniques are gaining attraction [33]. Combined with the software-based prefetching efforts [10], [32], hardware prefetching could provide greater benefits in the future.

VIII. Conclusion and Future Work

In this paper, we observe that "pipelining" and "blocking" in query processing are not well defined terms in the literature. We introduced the notion of UoT, which offers a clearer way to think about the data transfer mechanism between operators in a query processing pipeline. We propose an analytical model which includes a number of dimensions that impact query performance.

Our analytical model, as well as empirical evaluation for the Quickstep database system shows that the traditional "pipelining" and "non-pipelining" strategies are not very different w.r.t. query performance. Additionally, we looked at the memory consumption during query processing with varying UoT values. We found that the memory consumption of these two strategies is similar in many cases, and sometimes a non-pipelined execution strategy can have a lower memory overhead when it employs a bloom-filter based pruning technique.

For future work, we plan to revisit the assumptions made for "pipelining" in the context of new memory technologies involving complex storage hierarchies.

Acknowledgements

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA. Additional support was provided by the National Science Foundation (NSF) under grant OAC-1835446.

REFERENCES

[1] Hyrise. https://hpi.de/plattner/projects/hyrise.html. Accessed on 07/20/2018.

[2] Peloton - the self driving database management system. htts://pelotondb.io. (Accessed on 07/05/2018.

[3] TPC-H - Homepage. http://www.tpc.org/tpch/.

[4] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD Conference*, pages 967–980. ACM, 2008.

[5] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[6] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48. ACM, 2011.

[7] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237. www.cidrdb.org, 2005.

[8] C. Chasseur and J. M. Patel. Design and evaluation of storage organizations for read-optimized main memory databases. *PVLDB*, 6(13):1474–1485, 2013.

[9] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating progress of long running SQL queries. In *SIGMOD Conference*, pages 803–814. ACM, 2004.

[10] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116–127. IEEE Computer Society, 2004.

[11] T. Cruanes, B. Dageville, and B. Ghosh. Parallel SQL execution in oracle 10g. In *SIGMOD Conference*, pages 850–854. ACM, 2004.

[12] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. In *SIGMOD Conference*, pages 215–226. ACM, 2016.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[14] H. Deshmukh, H. Memisoglu, and J. M. Patel. Adaptive concurrent query execution framework for an analytical in-memory database system. In *BigData Congress*, pages 23–30. IEEE Computer Society, 2017.

[15] D. J. DeWitt, S. Ghandeharizadeh, and D. A. Schneider. A performance analysis of the gamma database machine. In *SIGMOD Conference*, pages 350–360. ACM Press, 1988.

[16] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[17] C. Freedman, E. Ismert, and P. Larson. Compilation in the microsoft SQL server hekaton engine. *IEEE Data Eng. Bull.*, 37(1):22–30, 2014.

[18] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *SIGMOD Conference*, pages 1603–1618. ACM, 2018.

[19] R. H. Gerber. *Dataflow Query Processing using Multiprocessor Hash-partitioned Algorithms*. PhD thesis, Madison, WI, USA, 1986. TR672.

[20] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[21] R. A. Hankins and J. M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, pages 417–428. Morgan Kaufmann, 2003.

[22] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD Conference*, pages 383–394. ACM, 2005.

[23] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72. ACM, 2007.

[25] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE Computer Society, 2011.

[26] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.

[27] W. Kim. Relational database systems. *ACM Comput. Surv.*, 11(3):187–211, 1979.

[28] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[29] K. Lee, A. C. König, V. R. Narasayya, B. Ding, S. Chaudhuri, B. Ellwein, A. Eksarevskiy, M. Kohli, J. Wyant, P. Prakash, R. V. Nehme, J. Li, and J. F. Naughton. Operator and query progress estimation in microsoft SQL server live query statistics. In *SIGMOD Conference*, pages 1753–1764. ACM, 2016.

[30] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD Conference*, pages 743–754. ACM, 2014.

[31] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, pages 829–840. ACM, 2005.

[32] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

[33] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2):35:1–35:35, 2016.

[34] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[35] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *TPCTC*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009.

[36] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

[37] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[38] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *VLDB*, pages 469–480. Morgan Kaufmann, 1990.

[39] V. Viswanathan. Disclosure of h/w prefetcher control on some intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, September 2014. (Accessed on 06/12/2018).

[40] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD Conference*, pages 1279–1294. ACM, 2016.

[41] Z. Yin, J. Sun, M. Li, J. Ekanayake, H. Lin, M. Friedman, J. A. Blakeley, C. A. Szyperski, and N. R. Devanur. Bubble execution: Resource-aware reliable analytics at cloud scale. *PVLDB*, 11(7):746–758, 2018.

[42] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel. Looking ahead makes query plans robust. *PVLDB*, 10(8):889–900, 2017.

[43] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *VLDB*, pages 723–734. ACM, 2007.

[44] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, pages 1349–1350. IEEE Computer Society, 2012.