ILMENAU UNIVERSITY OF TECHNOLOGY

Institut für Praktische Informatik und Medieninformatik
Fakultät für Informatik und Automatisierung
Fachgebiet Datenbanken und Informationssysteme

Master thesis

Achieving many-core scalability in Vectorwise

vorgelegt von:	Tim Gubner	
Matrikel:	44130	
Betreuer:	Prof. Dr.Ing.	Kai-Uwe Sattler
	Prof.	Peter Boncz

September 4, 2014

Zusammenfassung

Diese Masterarbeit erfüllt zwei Zwecke: 1. gibt sie einen Überblick über die Probleme, welche die Skalierbarkeit von Vectorwise auf Mehrprozessorsystemen einschränken und 2. stellt sie einen Ansatz zur Modernisierung der - im Datenbankbereich - vorherrschenden Technik der Intra-Query-Parallelisierung, dem Volcano-Modell-Parallelismus, dar.

Die Analyse der Probleme, welche die Skalierbarkeit von Vectorwise behindern, wurde größtenteils über die Menge der TPC-H Anfragen durchgeführt. Es wurde herausgefunden, dass die Hash-basierte Synchronisation und Neuverteilung der Daten der Producer-Threads hin zu den Consumer-Threads teil dieser Probleme ist. Weiterhin zählt das sequentielle Aufbauen der Hash-Tabelle beim HashJoin-Operator und der Reuse-Operator, welcher, da dieser komplett sequenziell abläuft, ein vorheriges Zusammenfügen und u.U. spätere Neupartitionierung der Datenströme erfordert, dazu. Weiterhin wurde aufgezeigt, dass Sperren in der E/A-Schicht ebenfalls zu diesen Problemen zählen. Ein generelles Problem, dass bei der Verarbeitung von Anfragen auftreten kann, ist eine ungleiche Verteilung der Arbeitslast, welche u.U. dynamisch während Datenbankanfragen auftritt. Der Volcano-Modell-Parallelismus bietet, aufgrund seiner statischen Natur, keine Möglichkeit dieses dynamische Auftreten auszugleichen. Es bestünde die Möglichkeit das Parallelismus-Modell zu ändern, was aber in einem bestehenden System zu einem enormen Anpassungsaufwand führt, da im Volcano-Modell die Parallelarbeit selbst ausgelagert ist und vor den Operatoren verborgen bleibt.

Diese Arbeit stellt einen Versuch zur Modernisierung des Volcano-Modell-Parallelismus dar. Es wurde gezeigt, dass dass die Hürden der Volcano-Herangehensweise überwältigt werden können. Der Ansatz hierfür war die Benutzung einer signifikant höheren Anzahl an Threads (im Vergleich zu der Anzahl an Prozessoreinheiten) und eine permanter Kontrolle der Threads durch einen User-Level-Scheduler, wobei dieser versuchte dynamisch auftretende Effekte zuvermeiden bzw. abzulindern. Dabei war der Auflaufkoordinierungs-Algorithmus (Scheduling-Algorithmus) daraufhin ausgerichtet alle Threads, welche Teile der Anfrage bearbeiten auf den gleichen Fortschritt zu bringen bzw. zurückliegende Threads zu bevorzugen oder ggf. zu beschleunigen.

Zusammenfassend kann gesagt werden, dass es möglich war die Probleme des statischen Volcano-Modell-Parallelismus auf einem effizienten Weg zu umgehen, was sich anhand der Messergebnisse, von teilweisen Messungen der Balance, die der User-Level-Scheduler erzielen konnte, bis hin zu TPC-H zeigen lässt.

Abstract

This thesis has two major purposes: (1) to analyze the issues hindering (many-core) scalability in Vectorwise and (2) to modernize the still dominant approach for intra-query parallelization: Volcano-model parallelism.

The analysis was conducted over the TPC-H queries which showed that hash-based (re-)distribution from producer threads to consumer threads is one of factors limiting scalability. Other issues limiting scalability were the sequential building of HashJoin's hash table and the Reuse operator which runs sequential and forces parallel stream to be joined before and forked afterwards. Further the survey showed that locking in the I/O layer can become a scalability issue. A general issue in parallel query evaluation is skew which may dynamically appear in query plans. The Volcano-model parallelism provides no way handling dynamic effects due to its static nature. It is possible to change the parallelism model, which - in an existing system - usually involves a big re-engineering overhead, because - in case of the Volcano-model parallelism - the parallelism is "encapsulated" and operators are kept unaware of parallelism.

This thesis provides an attempt to modernize the Volcano-model parallelism. It was shown that it is possible to overcome the problems of the Volcano-approach to parallelism by using significantly more threads than processors and scheduling these in order to reduce dynamic effects. This was reached by a scheduling algorithm which tries to keep all query evaluation threads at the same progress and boost threads lagging behind.

Summarizing it was possible to find an efficient way out of the static Volcano-model parallelism - as can be shown over microbenchmarks and the TPC-H benchmark.

Contents

1. Introduction

According to [Sut09] single-core CPU performance reached its zenith which forces applications to use parallelism in order to accelerate their computations. But not every parallelism model is efficient in every scenario. One would define parallelism to be efficient when the applciation is able to take advantage by using additional computing power. This can be referred to by using the term *scalability* in which the applications performance shall *scale* with the amount of resources used.

It is possible to introduce parallelism in multiple ways into a DBMS in order to accelerate query processing. Two popular methods are:

- *Inter query parallelism*, which means that multiple queries are allowed to run concurrently, and
- *Intra query parallelism* where a query is accelerated by using parallelism.

These two methods are completely orthogonal to each other and can be combined. Where the latter one (*intra query parallelism*) can be refined into

- *Inter operator* where different operators are executed concurrently and
- *Intra operator* where parallelism is introduced inside an operator.

Vectorwise is an analytical database system. It is known for efficiency by exploiting modern processor's features like instruction-level parallelism and excessive use of on-chip caches [BZN05]. Vectorwise utlizes

- Inter query parallelism by using at least one kernel thread per query (until a configurable limit) and
- Intra query parallelism by the following techniques:
  - Inter operator parallelism is introduced by a set of - so called - *exchange* operators [Gra94, p. 105] which parallelize their subtrees.
  - Further intra operator parallelism is exploited through the usage of SIMD[1] and pipeline parallelism inside the processor(s).

[1]Single instruction, multiple data

Note that the concept of *exchange* operators is the currently dominant approach used. For example it is used in Microsoft SQL Server [LBKN14, p. 11], Oracle [LBKN14, p. 8], Vertica [LFV⁺12, p. 1797] and indeed Vectorwise. The reason for this is that *exchange* operators "encapsulate" parallelism [Gra94, p. 102] in a way that original relational query operators (e.g. Join, Sort and Aggregation), are unaware of parallelism. In case these operators were implemented in a non-parallel way, they could automatically take advantage of parallelism without any modification of the operators itself. Hence introducing parallelism using *exchange* operators minimize software re-engineering cost. In contrast, recently developed database systems, such as HyPer [LBKN14] and BLU [RAB⁺13], argue against the concept of *exchange* operators and make each operator aware of parallelism. In the approach chosen by HyPer and BLU each operator divides the work into many small tasks and uses a queue to distribute tasks dynamically between the processors (hence task-based parallelism). Such an approach arguably can achieve

- better load-balance between the processors,
- better NUMA[2] locality and
- better workload adaption

compared to the *exchange* approach which partitiones work (for each thread) statically at query compilation time.

1.1. Motivation

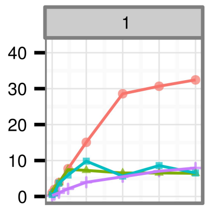


Figure 1.1.: Speedup gained by Hyper (red line) and Vectorwise (purple line) over TPC-H scale factor 100 on Nehalem EX from [LBKN14, p. 9]

Consider TPC-H Q1. Figure 1.1 visualizes the speedup gained by HyPer and Vectorwise over Q1 in TPC-H scale factor 100. It can be seen that Hyper, represented by the red line, shows a speedup of 30 when using 32 processors. While "Vectorwise has similar single-threaded performance as HyPer" [LBKN14, p. 8] its speedup gained, visible as the purple line, by using 32 processors is approximately 7. [LBKN14] relates the different speedups to the "use of the Volcano model for parallelizing queries in Vectorwise" [LBKN14, p. 8].

[2]Non uniform memory access

Considering these disadvantages of the *exchange* approach to parallelism, it would involve a lot of re-engineering to move from the *exchange* based parallelism to the task-based parallelism as it is used by HyPer.

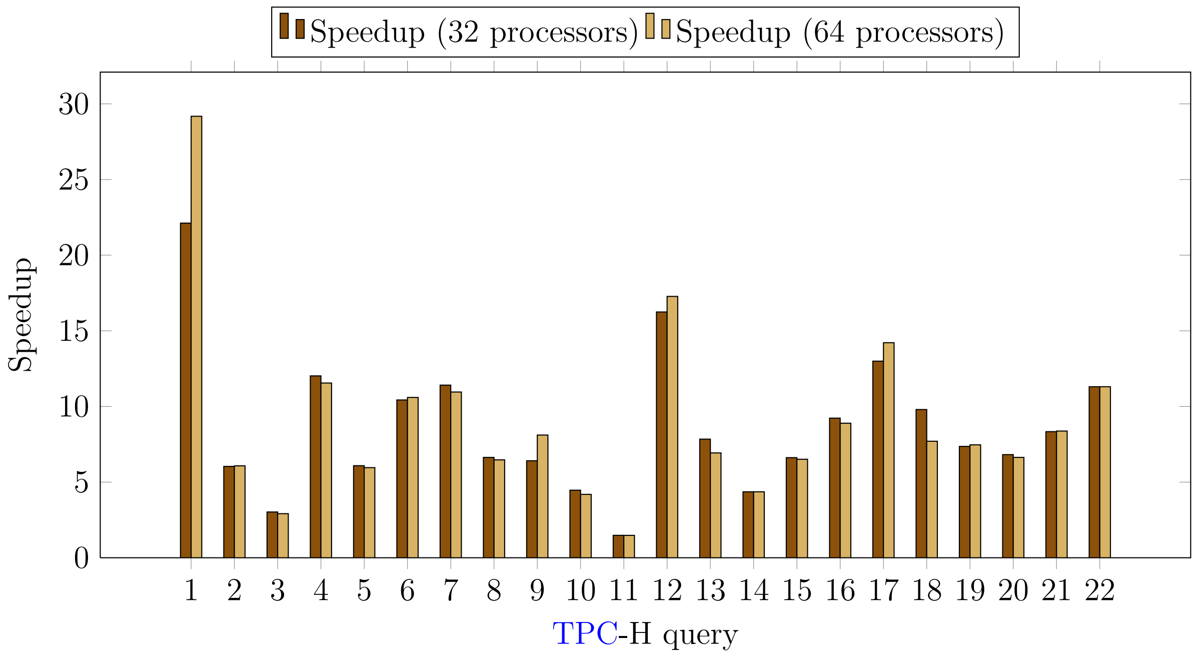


Figure 1.2.: Speedup gained for the TPC-H queries using 32 and 64 processors on scale factor 500

Going back to the scalability of Vectorwise which is visualized Figure 1.2. It visualizes the speedup using 32 processors and 64 processors for each TPC-H query when using scale factor 500. The best speedup (according to Figure 1.2) was achieved by Q1. It provides a speedup of ≈ 29 by using 64 processors, but it is obvious that Q1's response time does not scale linearly with the number of processors.

This thesis will provide reasons (here called "*scalabilty issues*") that substantially hinder the speedup of Vectorwise.

1.2. System details

All benchmarks (in this thesis) were measured on a single system. This system will be explained in the following.

NUMA: Using modern mainstream server hardware it is possible to build a system with 64 processors (4 sockets with a processor package hosting 16 processors each) and a large amount of main memory without requiring special hardware. This hardware uses a special kind of interconnection between the CPUs, because the ”bus-based, global-memory architecure of small systems does not scale because the shared bus quickly becomes a system bottleneck as [..] more processors are added” [ZB91, p. 1]. Thus these hardware systems use a different approach to efficiently support a high number of CPUs.

One possible approach is to trade ”the idea that all memory modules have the same access time” [Tan01, p. 511] for better scalabilty (i.e. a higher number of CPUs). Such systems are called NUMA. They have the important property that all CPUs can access one single address space, but accessing the memory might take different time. There are two types of memory accesses:

- There is *local memory access* and
- *remote memory access*

The difference is that it can be assumed that ”remote memory [access] is slower than access to local memory” [Tan01, p. 511]. These types result from the partitioning of main memory over all memory controllers. This is usually realized through using the memory banks proximite to the memory controller. When considering multi-core processors - each with its own memory controller - this forms regions where a memory controller can access the memory local to itself (its partition). Such a region is called *NUMA node*. This implies that on each NUMA node, there is local memory accessed by a local memory controller. Multiple local processors that have *local access* to the local memory and a network connection to allow remote processor (from other NUMA nodes) to access the local memory (*remote access*).

Hardware: Used was a four-socket mainboard using four AMD Opteron 6376 with 32 GiB memory attached to each socket, summing up to 256 GiB in total. Further on each socket sits a processor package hosting two dies. Each die has its own memory controller, a shared L3 cache with a capacity of 16 MiB and a link to the other die. Futhermore each die hosts four Piledriver modules. Further each Piledriver module hosts two cores which are sharing the instruction-fetcher and decoder, the L2 cache and the FPU[3]. Each die is considered a NUMA node because it has its own memory controller. One might assume that these 8 NUMA nodes are all connected i.e. to the other 7 (NUMA) nodes. This is clearly not the case. Figure 1.3 shows the link topology of all eight NUMA nodes where the NUMA nodes with odd numbers (o) are on the same socket as $o - 1$ e.g. 0 and 1; 2 and 3 are on one socket.

Furthermore this figure shows which NUMA nodes are connected: The NUMA node with even numbers are connected with all other nodes with even numbers (excluding themself).

[3]Floating point unit

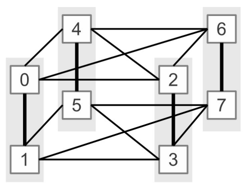


Figure 1.3.: Link topology [DGT13, p. 4]

Each even (NUMA) node is also connected with the other node on its die which has an odd number and all nodes with odd numbers are connected with all other nodes with odd numbers (excluding themself). Interpreting Figure 1.3 as a network graph, the shortest distance from one node to another can be one hop or two hops depending which node communicates with which.

Cache hierarchy: As stated by [Dre07, p. 16] AMD prefers to use a concept called *exclusive caches* where each cache (L1, L2, L3) holds a disjoint set of cache lines. When a cache line is evicited from a cache, then - according to [Dre07, p. 16] - it needs to evict a cache line from a cache one step closer to main memory or the main memory itself. For example, when a cache line is evicted from L1 cache, it needs to push the cache line into the L2 cache, which also needs to evict a cache line and so on until the main memory is reached. ”A possible advantage of an exclusive cache is that loading a new cache line only has to touch the L1d and not the L2, which could be faster” [Dre07, p. 16].

Cache coherency: In order to remain a consistent view of the memory (including the caches) a cache coherency protocol is needed. According to [AMD13b, p. 169] the AMD Opteron processors are using - in contrast to Intel Xeon - the MOESI[4] protocol for maintaining cache coherency. The MOESI protocol allows a cacheline to have one of the following five states:

- ”A cache line in the *modified* state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy” [AMD13b, p. 169].
- Each ”cache line in the *owned* state holds the most recent, correct copy of the data. The owned state is similar to the shared state [...]” [AMD13b, p. 169], but unlike ”the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state” [AMD13b, p. 169] where - according to [AMD13b, p. 169] - all other processors must hold the cache line in shared state.

[4]Modified Owned Exclusive Exclusive Invalid

- ”A cache line in the *exclusive* state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data” [AMD13b, p. 169].
- ”A cache line in the *shared* state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. If no other processor holds it in the owned state, then the copy in main memory is also the most recent” [AMD13b, p. 169].
- Each *Invalid* cache line do[es] not hold a valid copy of data, as stated by [AMD13b, p. 169]. Further ”valid copies of the data can be either in main memory or another processor cache.” [AMD13b, p. 169].

1.3. Vectorwise algebra

The query evaluation in Vectorwise is built upon operators, which reassemble a ”rather standard relational algebra” [BZN05, p. 231]. These operators form a tree (further referred to as QEPs[5]) and ”operate in a demand-driven schema” [Ani10, p. 13].

In the following the operators, used in this thesis, will be described:

- The *Project* operator allows introducing new attributes, renaming attributes and to filter entire attributes from a dataflow.
- The *Select* operator allows to filter out tuples from a dataflow that do not match a predicate.
- The *Aggr* operator which allows ”to represent a set of items by a single value or to classify items into groups and determine one value per group” whereas the *OrdAggr* is an optimization of *Aggr* for the case when ”group-members will arrive right after each other” [BZN05, p. 232] in the dataflow.
- The *HashJoin* (*HashJoinN*) operator builds a hash table from its left input dataflow and afterwards probes tuples of the right input dataflow through the hash table. For each match in the hash table an tuple in the output dataflow is created.
- *HashJoin01* operator is an optimization of the *HashJoinN* operator for zero or one match in the hash table.
- The *MergeJoin* operator resembles the merge part of the Sort-Merge-Join algorithm, hence it has two input dataflows.

[5]Query execution plans

- The *Sort* sorts the dataflow.
- The *TopN* operator allos to determine the top-n tuples by a given sorting criteria.
- The *Reuse* operator provides the functionality to break out of the operator tree. It allows to store tuples and returns the input dataflow while other instances of the *Reuse* operator can read these stored tuples.
- The *As* operator allows to rename the input dataflow's table qualifier.
- The *Xchg* operator resemble the *exchange* operator from [Gra94]. This operator provides a synchronization point between m producer threads and n consumer threads. It parallelizes its subtree using m threads and buffers the results from each thread in order to be consumed by the n consumer threads. Note when a subtree is already parallelized using a *Xchg* operator it will not be parallelized again, instead it redistributes data between the producer and consumer threads.
- Based on the *Xchg* operator the *XchgUnion* and the *XchgHashSplit* operator can be defined: The *XchgUnion* is a special case of the *Xchg* operator with only one consumer thread. The *XchgHashSplit* is another special case in which the consumer thread of each tuple is determined via a hash function.
- Last but not least, the *MScan* operator allows to scan a relation which is on-the-fly merged with deltas gained from the modifcation of the dataset and converted into a dataflow.

1.4. Contribution

This thesis will contribute an analysis Vectorwise's weakpoints with regard to many-core scalability and will further present feasible solutions how to overcome these weakpoints.

One of these weak points is the static Volcano-model parallelism which cannot react to dynamic workload changes as they may be introduced by skew. In this work the assumption is that it is not possible to deviate (much) from the *exchange* approach to parallelism while still being able to achieve a similar efficiency as obtained by recent systems like Hyper or BLU. As such, one of the main contributions in this thesis is a design for *user-level* thread scheduling, where many threads, running (parallelism unaware) database operators, are scheduled on a significantly smaller number of processors. Each thread is scheduled with the goal to make all threads, working on the same query operator, finish (almost) simultaneously, by taking the into *query progress* account. Thus achieving load-balance and giving the possiblity to adapt to workload changes. Further it will be argued that the proposed approach should be complemented with limited awareness of parallelism to implement NUMA optimizations. As such the work in this thesis is an attempt to modernize the *exchange* approach to parallelism in order to allow existing systems to flourish on modern many-core machines.

1.5. Structure

This document is structured into the eight chapters:

The following chapter *Survey* provides an analysis of scalability issues in Vectorwise. It starts with Section 2.1 which describes which hardware features hinder scalability. It is followed by an analysis of used parallelism model in Section 2.2. The subsequent Section 2.3 provides an analysis of the sequential portions in the TPC-H queries that - according to Amdahl's law - have negative impact on scalability. This analysis itself is involves an analysis of sequential phases (Section 2.3.1), sequential query parts (Section 2.3.2), of the *HashJoin* (Section 2.3.3) and *Reuse* operator (Section 2.3.4) as well as an analysis of the overhead involved by locking (Section 2.3.5). Further the influence of memory locality on the query response times is explained in Section 2.4, which is structured into an analysis of the memory locality inside the *MScan* operator (Section 2.4.1) and *HashJoin* operator (Section 2.4.2). This is followed by Section 2.5 which describes the influence of skew on scalability.

After that an overview about related research will be given in the Chapter 3. It is split into an overview about state-of-the-art parallelism models for query evaluation (Section 3.1), followed by related work about user-level scheduling, which is described in Section 3.2, and by Section 3.3 which gives an overview about schemes for estimating the progress of a running query.

Chapter 4 will introduce a model which allows to estimate the progress of a query or even a part of a query. It starts with the introduction of the used approach, which is described in Section 4.1 and is followed by Section 4.2 which defines of query progress, as it will be used in this thesis, provides a set of conventions that will be used in the following sections. The consequent sections define the progress over disjoint sets of operators, starting with Section 4.4 which defines the progress for the set of *scan operators*. In Section 4.5 defines the progress for the set of *streaming operators*, followed by Section 4.6 which defines the progress for *blocking operators* and Section 4.7 which defines the progress for *buffering operators*.

At the end of the Chapter 4 - in Section 4.9 - the presented model will be evaluated regarding different properties. At first the linearity of the estimated progress is analyzed in Section 4.9.1, then in Section 4.9.2 will be proven that the progress is monotonic.

The subsequent Chapter 5 describes an approach for making static Volcano-model parallelism dynamic by using overallocation and load-balancing. Chapter 5 starts an explaination of the used approach (Section 5.1). This approach is further structured into four sections: Section 5.1.1 explains how to calculate the time to completion from the estimated progress, followed by a - in Section 5.1.2 described - workload metric based on the time to completion. The subsequent Section 5.1.3 explains how a query is split into multiple parts, which is further used for implementing the scheduling algorithm. The following Section 5.1.4 highlights the ideas that were used in order to implemented scheduling algorithm. Afterwards the implementation of the user-level scheduling is described in Section 5.2, which is further partitioned into the following six sections: First the integration of overallocation into the rewriter is explained in Section 5.2.1. The subsequent Section 5.2.2 describes how the thread synchronization using mutexes and condition variables is implemented in cooperation with the user-level scheduler. It is followed by Section 5.2.3 which describes a low-level abstraction of switching from one thread's execution context to another. Based on this, the following Section 5.2.4, explains how a context switch is realized in the user-level scheduler. In the consequent Section 5.2.5 the used scheduling algorithm will be described, followed by a description of how time quantums are used in a cooperative multithreading environment to extend cooperative multithreading with a spice of preemption (Section 5.2.6). After the implementation details have been described, the implemented user-level scheduler will be evaluated in Section 5.3, starting with a short analysis of the achieved balance using a small set of short-running queries (Section 5.3.2). Further in Section 5.3.2 a microbenchmark will demonstrate that the combination of overallocation and user-level scheduling is able to achieve load-balance in the presence of skew. This is followed the the evaluation of this combination over the set of TPC-H queries (Section 5.3.3).

The subsequent Chapter 6 summarizes the knownledge gained from the previous chapters. This is split into three part, starting with Section 6.1 which describes the conclusions that can be drawn from the survey of scalability issues done in Chapter 2. The subsequent Section 6.2 summarizes the, in Section 6.2 explained, query progress estimation, followed Section 6.3 which provides a summary of the user-level scheduling as it was described in Chapter 5.

Lat but not least the Chapter 7 gives insight about future extensions of the analysis about the scalability issues (Section 7.1), the query progress estimation (Section 7.2) and the user-level scheduler (Section 7.3).

2. Survey

In order to find out which problems are hindering scalability an analysis has been done, therefore the TPC-H benchmark (using scale factor 500) was run on the system described in Section 1.2.

Scenario: In order to analyze the scalability issues the survey was restricted to the following scenario:

- The Vectorwise server process (*x100_server*) is the only running process on the used machine.
- Furthermore all queries are evaluated in-memory only.
- The data stored is not modified during queries.
- CPU frequency boost was disabled for the measurement, because frequency boost is switched off when many cores are used, linear scaleup would be impossible by definition.
- Further in case of only using 32 processors, only the processors with even number where used (one per Piledriver module), because when two threads that shared the same Piledriver module would be used, the fact that inside a Piledriver module computation resources are shared would cause interference between them, making linear scaling impossible by definition.

2.1. Hardware features

The used hardware provides features which make it impossible to achieve linear scalability (of the query response time with the number of utilized processors). These features will be described and their implications, on scalability, analyzed in the following paragraphs of this section.

Frequency boost: One is the frequency boost which is activated when only a few processors are activly running on an AMD Opteron processor package. This behaviour is called *Turbo mode* and kicks in, when ≤ 8 processors are active. This feature boosts, according to [AMD13a, p. 1], the standard frequency (when all 16 processors are running) from 2.3 GHz up to a frequency of 3.2 GHz.

Shared processor components: The processors on the used AMD Opteron (processor) packages are not completely independent, because they share components. Each (of the used) AMD Opteron packages is partitioned into dies sharing a memory controller and the L3 cache. Each die hosts four ”Piledriver” modules of which each module share

- one instruction-fetch & decode unit,
- one FPU
- and one *L2 cache*.

Both features make it theoretically impossible to achieve linear scalability, because using more cores may slow down the whole processing. Further the frequency boost was disabled and the frequency of all processors was fixed to the standard frequency by disabling power management features.

2.2. Parallelism model

In this section the parallelism model used in Vectorwise will be discussed, which was implemented in the frame of [Ani10]. The used model is the Volcano-model, first described in [Gra94], which allows to keep most operators unaware of parallelism using parallelization operators. These are introduced through a cost-based optimizer.

2.2.1. Chosen parallelism

This cost-based optimizer chooses the parallelized QEP and determines the degree of parallelism used. Sometimes the optimizer limits the chosen degree of parallelism below the number of available processors. This may happen e.g. due to concurrently running queries or, in the case of only a single query, due to certain overheads that grow with the chosen level of parallelism. Figure 2.1 shows the maximal degree of parallelism given by the optimizer. Further it visualizes the the speedup gained by using the parallelized QEP compared to the sequential QEP. It can be seen that even with a high degree of parallelism (e.g. 64) the average speedup is ≈ 10. So that, in the measured TPC-H workload, the

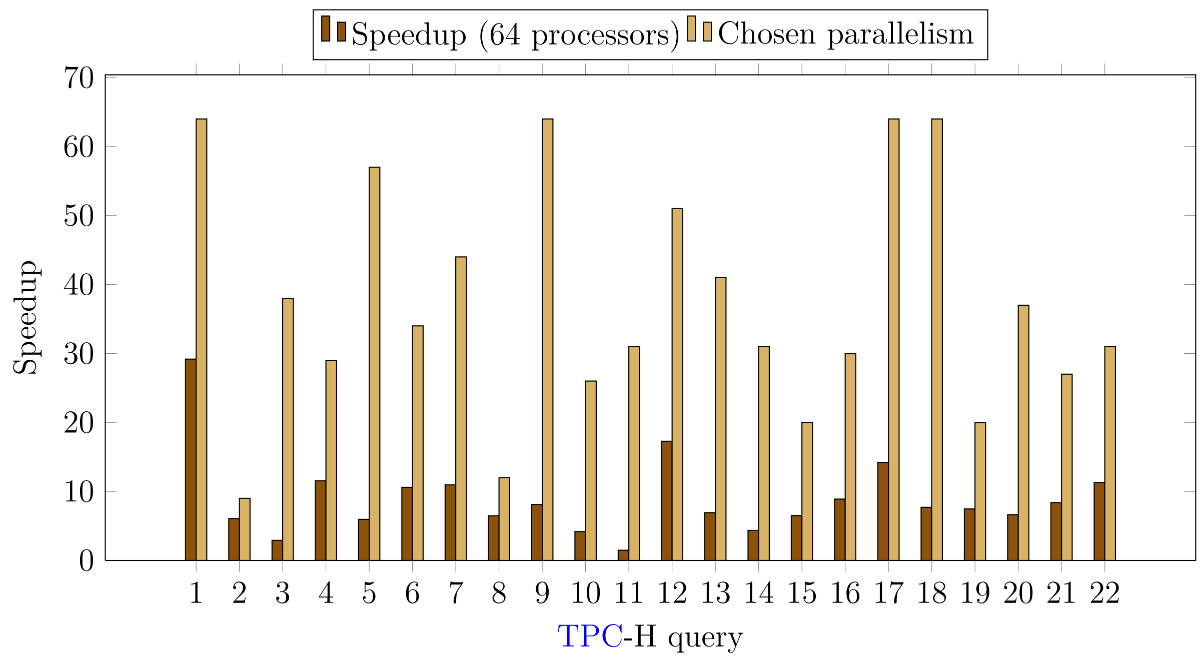


Figure 2.1.: Speedup gained for the TPC-H queries (on 64 processors)

scalability is not limited by the optimizer. But it can be seen in Figure 2.1 that most queries cannot exploit the maximal degree of parallelism provided by the hardware.

2.2.2. *Xchg* operator

In Vectorwise intra-query parallelism is introduced using a QEP rewriting rule which introduces *Xchg* operators [Ani10]. Each *Xchg* operator provides a data redistribution between m producer and n consumer threads. As implemented in Vectorwise it uses buffers to temporarly store the data (from the vectors). These buffers can be seen as a memory region shared by producer and consumer threads over which they (producer threads and consumer threads) communicate.

Memory locality: The buffers are shared by multiple threads. This makes these buffers are a potential subject to memory locality! Further buffers are firstly written by the producers. Applying Linux' first-touch policy to that implies that the buffers are residing on the NUMA node where the producer thread was running on. Based on the semantics of a *Xchg* operator there can be made serveral assumptions about the memory placement.

- In the case of a *Xchg* with one producer thread (i.e. $m = 1$) this might be one NUMA node because only one thread writes into the buffers and making them local to the NUMA node that made the first write to the buffer. This NUMA node might become a potential bottleneck for memory access!

- When having more than one producer thread the buffers are residing on random NUMA nodes, because there cannot be made any statement which thread will write first into a buffer.

Increasing effort: At some point producer threads and consumer threads have to find a buffer to write to (producer thread) or read from (consumer thread). This uses linear search to find a buffer matching the producer or consumer thread's requirements and will not scale for a large number of threads. The cost of the linear search could be amortized at the expense of memory consumption i.e. using larger buffers.

2.2.3. *XchgHashSplit* operator

The *XchgHashSplit* operator is a special-case of the *Xchg* operator and provides a hash-based data redistribution of m input/producer threads to n output/consumer threads.

Memory consumption: In order to achieve that it needs (theoretically) at least one shared buffer for each producer and consumer thread. Practically, to amortize locking overhead, it uses at least 2 per producer-consumer pair. This leads to at least $2 \cdot m \cdot n$ Buffers.

Thundering herd: In case the buffers for a consumer are full or empty the accessing thread waits on a condition variable (*producer_cond*, *consumer_cond*). But the wakeup (i.e. signalling the condition variable) is triggered as a broadcast which is forcing all threads to reacquire the mutex again. [1] This leads to contention of the mutex.

After reacquiring the mutex each thread starts looking for its buffer and, in the worst-case, checks all $(2 \cdot n \cdot m)$ buffers. Since all producer or consumer threads got woken up by the condition variable this worst-cast happens often, because only a few succeed finding a buffer specifically addressed to them. The ones that did not succeed finding a buffer are put to sleep again.

2.2.4. Case study: *XchgHashSplit* operators in Q21

It can be shown in an experiment that the *XchgHashSplit* operator is a scalability issue. This experiment uses a slightly modified version of Vectorwise with the following modification:

[1] According to the pthreads specification [Gro14] waiting on a condition variable only be done in combination with a mutex. The general pattern is explained in Section 5.2.2.

- The rewriting rule which introduces parallelism was changed in order to introduce *Xchg* operators with a higher degree of parallelism into the QEP.
- Further a patch that works around the - previously mentioned - thundering herd problem was applied which only wakes specific threads *Xchg* operator and not all.

The changed rewriting rule leads to parallelized QEPs of Q21, as visualized in Figure 2.2 where n is the degree of parallelim. Further it can be seen that this QEP contains five *XchgHashSplit* operators where each *XchgHashSplit* operator has an index i in its superscript and the number of producer threads and consumer threads in its subscript. Note that in this experiment the number of producer and consumer threads is both n.

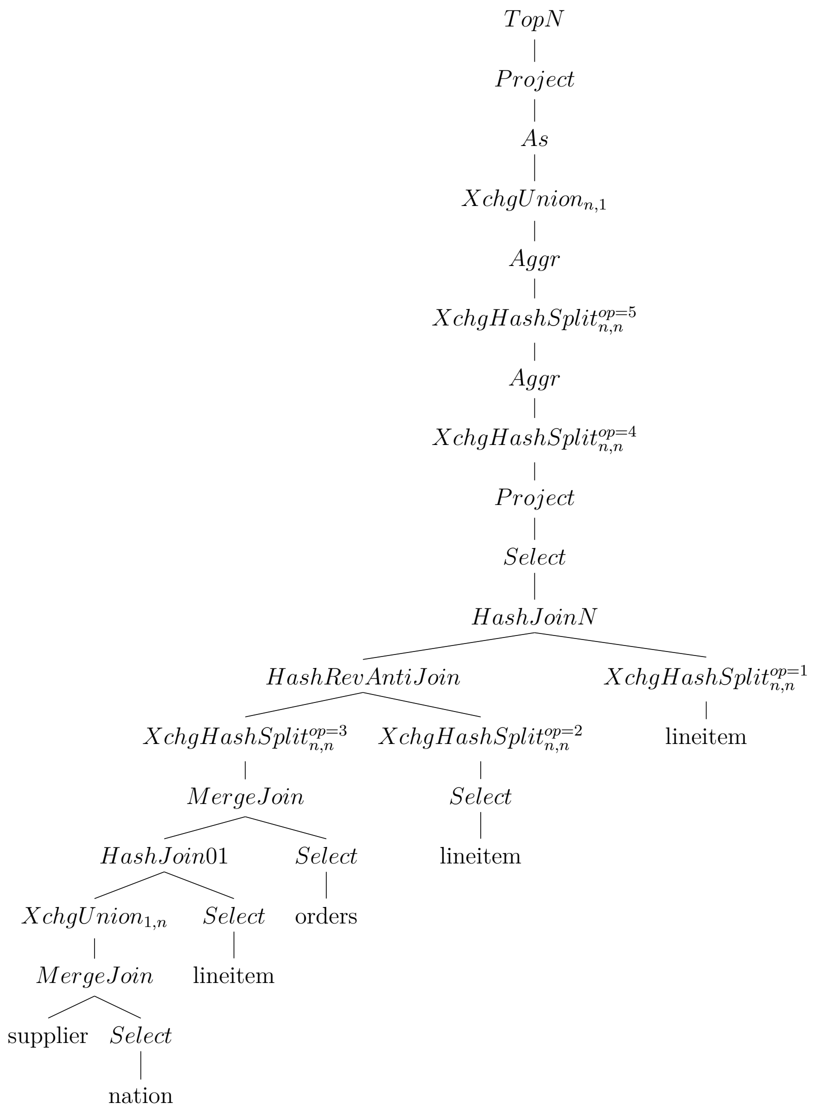


Figure 2.2.: QEPs of parallelized Q21

In general *XchgHashSplit* operator has exactly one mutex that protects the operator's shared state against concurrent access. Further the following different mutex times for

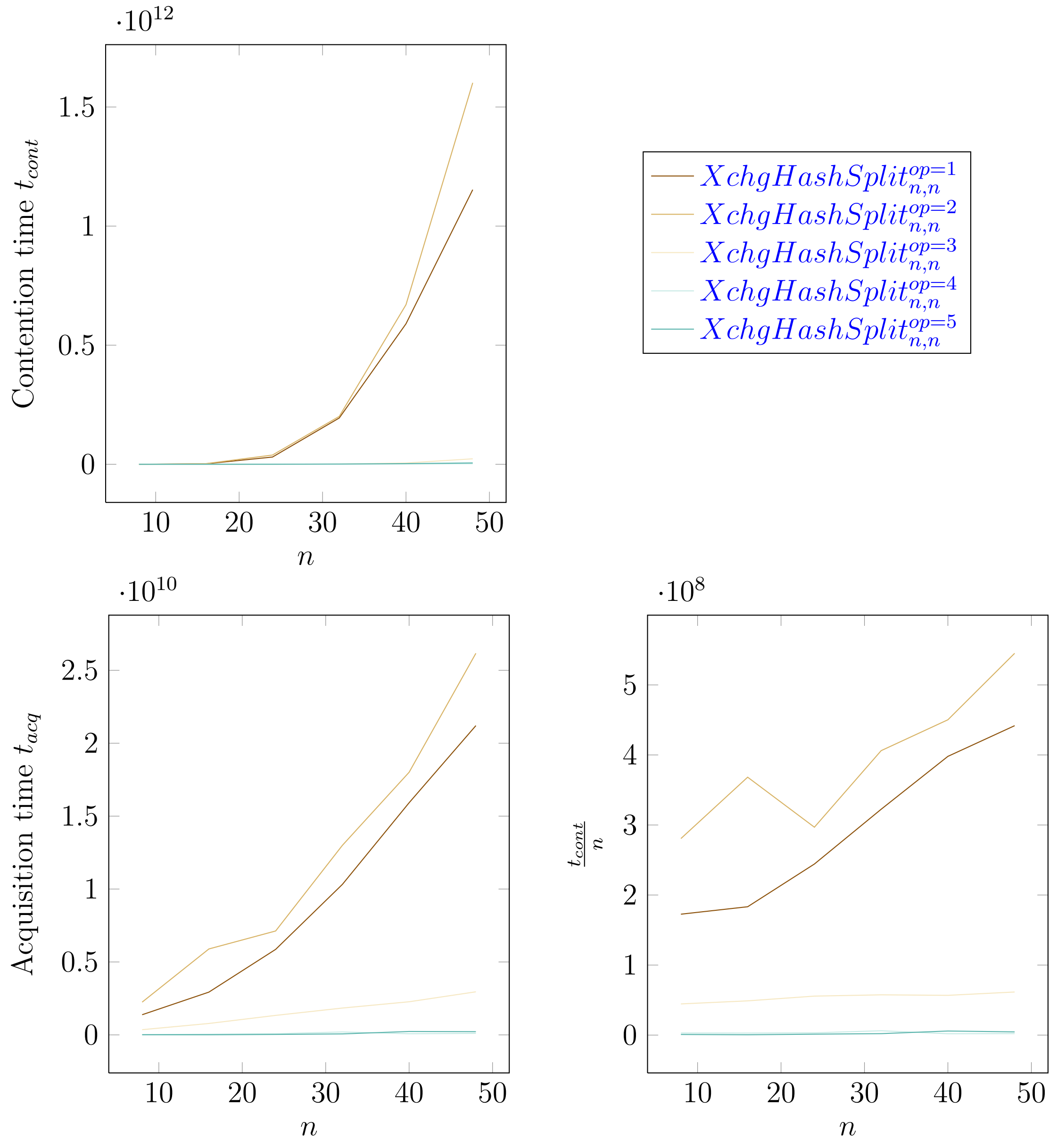


Figure 2.3.: t_{cont}, t_{acq} and $\frac{t_{acq}}{n}$ for each $XchgHashSplit^{op=i}_{n,n}$ from Q21

each *XchgHashSplit* operator will be analyzed:

- The length section of code protected by the mutex against concurrent access, the - so called - critical section - was measured as t_{cont}. Note that t_{cont} defines the sum of all critical section protected by operator's mutex. Further t_{cont} is measured in cycles.
- A mutex allows only one thread to succeed entering the critical section at a time and other threads trying to enter that critical section have wait until a thread leaves the critical section. The sum of all these waiting times is measured as t_{wait} in cycles.

Figure 2.3 shows t_{acq}, t_{wait} and $\frac{t_{wait}}{n}$ with a degree of parallelism $n \in \{8,\ 16,\ 24,\ 32,\ 40,\ 48\}$. In general it can be seen that the operators $XchgHashSplit_{n,n}^{op=3}$, $XchgHashSplit_{n,n}^{op=4}$ and $XchgHashSplit_{n,n}^{op=5}$ show no considerable times in the plots in Figure 2.3.

Table 2.1.: Tuples flown through each $XchgHashSplit_{n,n}^{op=i}$

i	Number of tuples	Number of buffers
1	$4.93 \cdot 10^7$	752.09
2	$3.12 \cdot 10^7$	475.52
3	$1.14 \cdot 10^6$	17.47
4	$1.17 \cdot 10^5$	1.78
5	$6.08 \cdot 10^4$	0.93

Contention time: In the t_{cont} plot it can be seen that in 2 of 5 *XchgHashSplit* operators - more precisely $XchgHashSplit_{n,n}^{op=1}$ and $XchgHashSplit_{n,n}^{op=2}$ - t_{cont} scales super-linearly in relation to n. Further in Table 2.1, which shows the number of tuples flown through each *XchgHashSplit* operator, these two operators are the ones with the highest number of tuples flown through. This moves the cost-center away from startup and teardown i.e. startup and teardown mutex contention is getting less relevant. In the *Xchg* operator the mutex is acquired during query execution everytime a producer or consumer thread needs to find a buffer. It can be shown that finding a buffer using linear search has a time complexity $\Theta(n^2)$ in average case: Let the number of buffers used be b. Further let the data be distributed from $m = n$ producer threads to n consumer threads, because only $XchgHashSplit_{n,n}$ are considered. This leads to

$$b = 2 \cdot n \cdot m = 2 \cdot n^2$$

Each search for a buffer, which must be done on both (producer and consumer) sides, is implemented as linear search over all buffers b, leading to an average case number of comparisons of

$$\frac{\sum_{k=1}^{b} k}{b} = \frac{b+1}{2} = \frac{2 \cdot n^2 + 1}{2} = n^2 + \frac{1}{2}$$

which is leading to a time complexity of

$$\Theta(b) = \Theta(n^2)$$

in the average case.

This search to be done for each producer and consumer thread when a new buffer is needed for temporarely materializing tuples or reading tuples from a buffer. Further assuming that the average length of the critical section is dominated by the search for a buffer. This leads to a theoretical mutex contention time (in the average case) of

$$t_{cont,theoretical} = \Theta((2 \cdot n) \cdot (n^2 + \frac{1}{2})) = \Theta(n^3 + n)$$

which is a cubic function.

In reality this model is only a coarse approximation for many reasons:

- Buffers may reside on remote NUMA nodes which increases memory access latency.
- Further caches is not considered in this approximation.

Given that the contention time scales super-linear with the degree of parallelism and that - from the sense of mutual exclusion - only one thread can execute the critical section at a time, the contention time of a mutex is a purely sequential fraction. Hence it is limiting the speedup according to Amdahl's law using a sequential fraction that increases with the number of threads.

In case the mutex is already contented threads trying to access the critical section have to wait. This implies that a longer critical section will force threads to wait longer, in the contented case.

Acquisition time: Further a longer contention time will also increase the chance to wait for acessing the critical section. Thus the sum of these waiting times - t_{acq} - may also increase. In Figure 2.3 can be seen that t_{acq} increases with the number of cores. This implies that the mutex must be contented in most of the cases where a thread wants to acquire the mutex. Further this waiting time could also be seen a sequential fraction, because a waiting thread cannot make progress and in the given scenario there is no other thread that could be efficiently be scheduled on this processor, because the other threads are already running on their processors. This could be done by calculating the average waiting time for one thread, as it was done in Figure 2.3, as $\frac{t_{acq}}{n}$.

2.2.5. Summary

In general it can be summarized that locking can become a scalability issue in *Xchg* operators. Furthermore there is a need for an efficient solution for finding buffers in the *Xchg* operator in order to damp the problems caused by locking.

Lock-free *Xchg* operator: In Section 2.2.3 it was shown that *Xchg* operators can behave very badly in given scenarios. In the case of the *XchgHashSplit* the time waiting on the mutex and the time spent in a critical section was problematic in terms of many-core scalability. One way out of this may be a lock-free *Xchg* operator. In [TFB+11] it has been shown that with a few restrictions it is possible to build a high-performance lock-free n-producer-m-consumer queue. This would be a nice fit for *Xchg* operator's implementation where such a queue is implemented using locking!

Further one problem occurs, how to synchronize faster/slower producers and consumer efficiently while not waiting for a potentially too long period? That will remain an open problem for future research.

2.3. Sequential fraction

Sequential percentages in parallel programs can substancially limit scalability, but it is impossible to completely eleminate these due to synchronization, setup and teardown overhead.

Amdahl's law: Equation (2.1) shows the formula known as Amdahl's law (as stated by [HM08, p. 34]) where f is the fraction of program's time that is parallelizable over m processors, hence $1 - f$ is the sequential fraction of the program.

$$S := \frac{1}{(1-f) + \frac{f}{m}} \tag{2.1}$$

Further "Amdahls law states that if a portion of a computation,f , can be improved by a factor m, and the other portion cannot be improved, then the portion that cannot be improved will quickly dominate the performance, and further improvement of the improvable portion will have little effect." [SC09, p. 184]

Assuming an infite number of processors used for parallelizing the maximal speedup is bound by

$$\lim_{m\to\infty} S = \lim_{m\to\infty} \frac{1}{(1-f) + \underbrace{\frac{f}{m}}_{=0}} = \frac{1}{1-f}$$

Figure 2.4 shows the speedup, according to Amdahl's law, for different fractions of parallelizable work f where the number of processors runs from 2 to 128. It can be seen that even a sequential fraction of 5% ($f = 0.95$) is limiting the scalability to the speedup of ≈ 10.

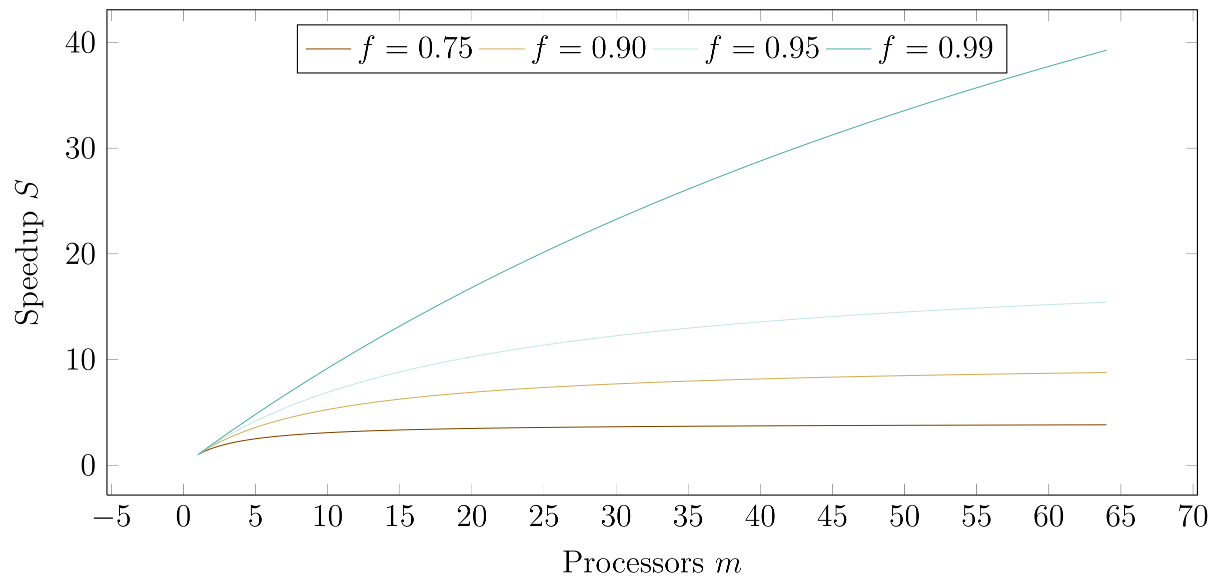


Figure 2.4.: Speedup according to Amdahl's law with differing parallelizable fraction f

2.3.1. Sequential phases

Figure 2.5 visualizes the architecture of the Vectorwise DBMS. It can be seen that no SQL queries reach Vectorwise only *query plans* reach Vectorwise and go through a predefined process, which is explained in the following:

- First the query (which is a textual representation of the QEP) is *parsed* into a tree structure (referred as QEP).
- Then the QEP goes through the *rewriting* process in which rewriting rules like parallelization may be applied (*Rewriter*).
- After rewriting on the basis of the QEP in the *Builder* a physical operator tree is constructed.
- This physical operator tree is then used for the *Query Execution (Engine)*.
- after query execution completed the allocated resources are *freed*.

Note that all these phases are run sequential and therefore represent a given sequential part limiting scalability (regarding Amdahl's law).

It was found out that the length of the *rewriting* phase and the phase in which the physical operator tree is *built* scale with the number of processors (here n) and start to become problematic with $n \geq 64$.

Rewriting: The most time consuming part of *rewriting* process is the parallelization rule which in the worst-case tries all parallelism levels between 2 and n. This worst-case

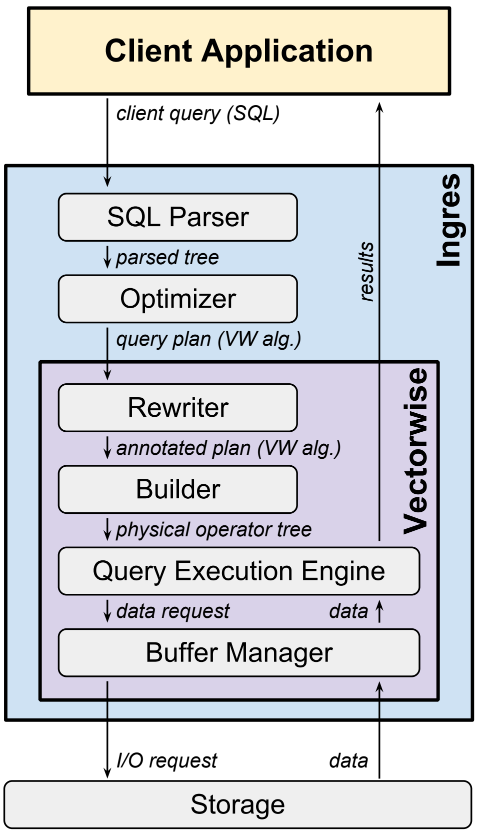


Figure 2.5.: Vectorwise DBMS architecture [CI12, p. 4]

happens everytime when a scalable QEP is detected i.e. when the cost of the QEP has its minimum with using all n processors. In this case all possible solutions for all processors starting with 2 until 64 were examined.

Build: After rewriting the QEP is finished the physical operator tree is *built* which builds any operator in the QEP recursively. In the Vectorwise system, building an operator means instantiating and initializing an operator object, which includes allocating and initializing its memory structures. Furthermore when a QEP is parallelized it has to build the whole parallel QEP n times, where n is the level of parallelism used.

Considering the example QEP in Figure 2.6, which consists of 7 operators, in sum 322 physical operators have to be built whereas Table 2.2 visualizes the calculation.

This means that the more parallelism added, the more expensive its get to build all these operators.

Sequential phases: According to Amdahl's law sequential percentages are a scalability bottleneck and the described sequential running phases are part of such sequential percentages. Figure 2.7 visualizes the fraction of each phase to the query response time.

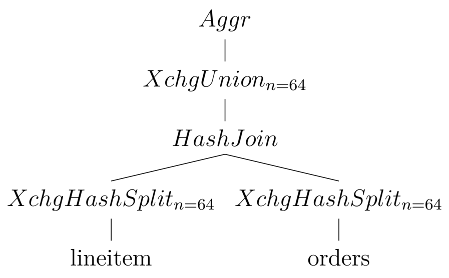


Figure 2.6.: Example QEP to build

Table 2.2.: Example calculation of the operators built

Operator	Count
Aggr	1
XchgUnion	1
HashJoin	64
left *XchgHashSplit*	64
Scan lineitem	64
right *XchgHashSplit*	64
Scan orders	64
$\sum$	322

Whereas *Not covered* means that the percentage shown was not included in the profile information gathered by Vectorwise where

$$not\ covered = total - (scan - parse - rewrite - build - execution - profile - free).$$

Note that in extreme cases, for example in Q10 only $\approx 70\%$ for the whole query response time are caused by *query execution*. Other extreme examples are

- Q4, where 80%, and
- Q14, where 70%, of the query response time is spent on executing the query.

2.3.2. Sequential query parts

Most parallelizable queries (even Q1) have one or more operators at the top-level of the QEP which are not parallelized at all. In case of Q1 these operators aggregate the partial results from the parallel subtrees, apply projections and sort the results. These top-level operators are executed sequentially which hinders scalability by introducing sequential work.

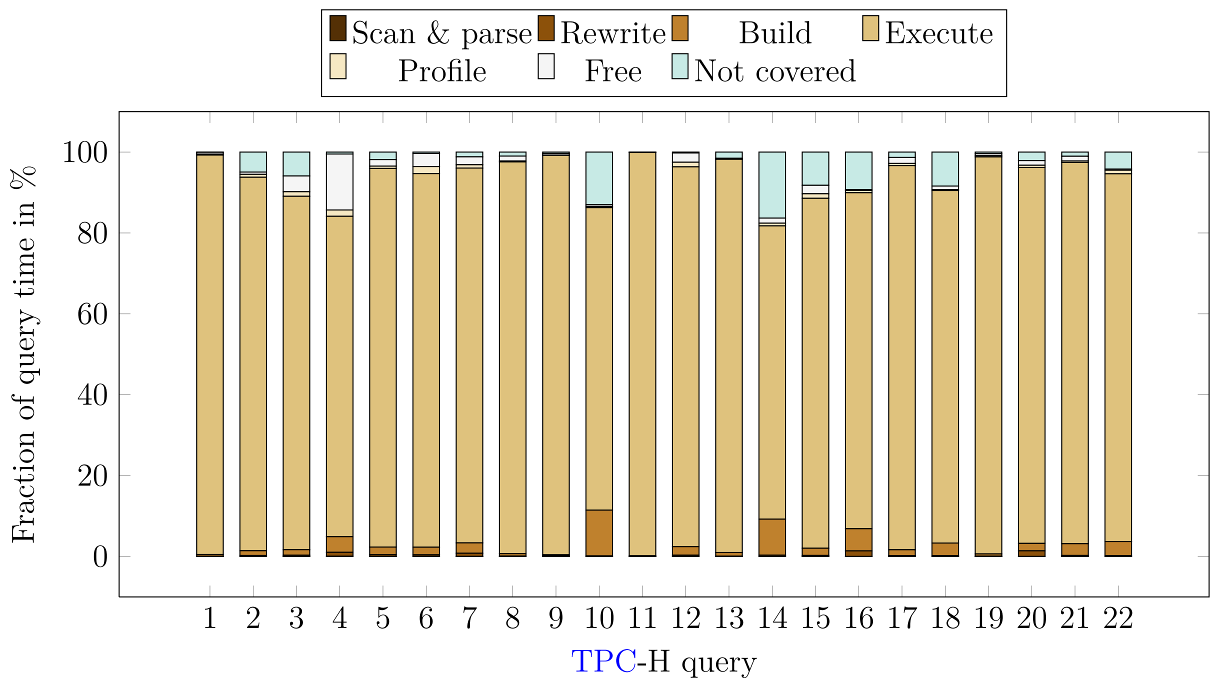


Figure 2.7.: Time spent in sequential phases in % of query time (on 64 processors)

Further the *Reuse* operator was not parallelizable (in the used version of Vectorwise) and forces the streams of the lower subtree to be joined and afterwards to be partitioned again. This obviously introduces sequential work and synchronization overhead where both is limititing scalability.

Another reason for sequential parts of the QEP is that depending on the cost model of the parallelism rewriter, for which it may appear cheaper to share a scanned relation instead of scanning it twice.

Figure 2.8 visualizes importance of the previously mentioned problems as a percentage of the query time (on 64 processors).

Figure 2.8 also shows that the sequential operators at the top of the QEP is a scalability killer for Q11 where $\approx 97\%$ of the query time is spent on doing sequential work while only a small part of the QEP was parallelized.

Further it can be seen that Q2, Q15 and Q17 spent more than 10% of their query time in the *Reuse* operator which therefore introduces a considerable sequential percentage.

The cost-based sequentially scanned relations which are shared afterwards may be a problem for very high scalability scenarios, but in the analyzed TPC-H queries it is a minor scalability issue because only $< 1\%$ of query time is spent there.

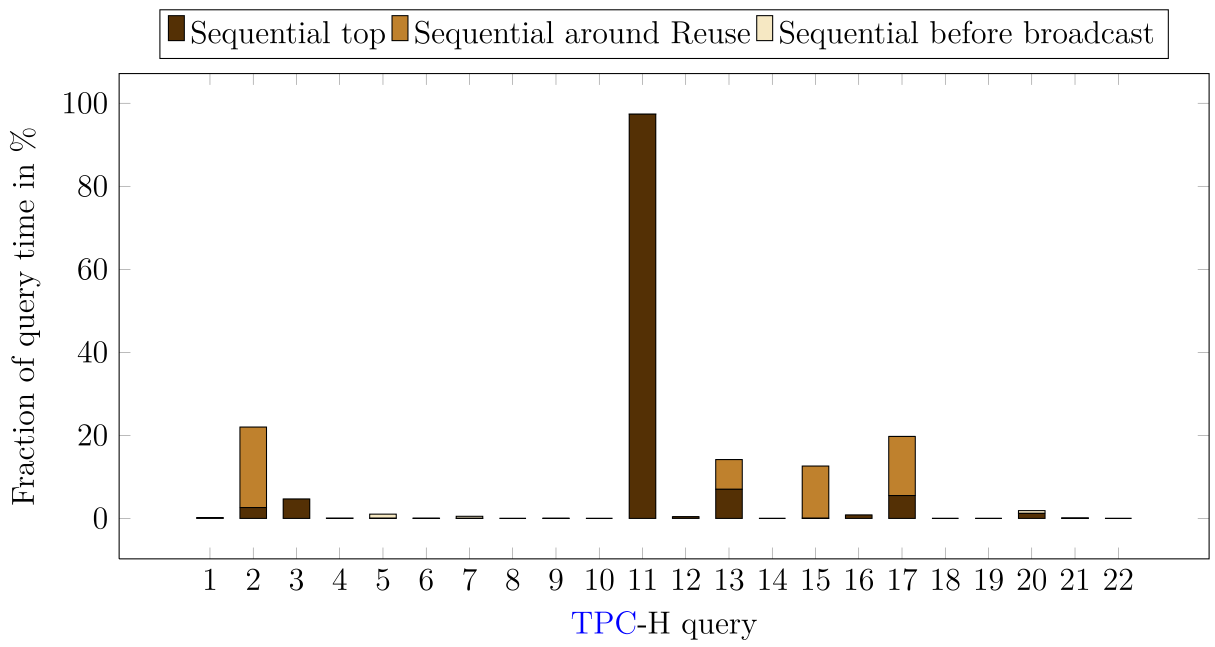


Figure 2.8.: Time spent in sequential parts as % of query time (on 64 processors)

2.3.3. *HashJoin* operator

In Vectorwise *HashJoin* operators can be parallelized in two ways:

- As independent *HashJoin* operators with (*XchgHashSplit*) on both sides, in which each *HashJoin* operator can process data independent of the *HashJoin* operators run by the other threads.
- The other way of parallelizing *HashJoin* operators is using a shared VHT[2] where all threads with a same build side wait until the VHT is constructed and - after the building has finished - share the same data structure. [3]

Shared VHT: In the Vectorwise system a shared VHT *HashJoin* involves one thread that builds the VHT. After the building has been completed multiple threads can probe in parallel using the same VHT. Note that sharing the VHT built by one thread and - according to Linux' first touch policy - assigned to one NUMA is very likely to become a bottleneck for memory access.

Table 2.3 shows the time (as a percentage of the whole queries response time) spent in building the VHT sequentially. Note that queries without a shared VHT *HashJoin* or a percentage below 0.1 % are not listed. It can be seen that for Q18 $\approx$ 44% of the query

[2]Vectorized hash table
[3]Note that shared VHT HashJoins are a departure of the pure Volcano-model parallelism, because the synchronization to wait for the shared VHT to be built introduces limited awareness of parallelism.

time is spent on building the shared VHT sequentially. A less extreme example is Q3 where ≈ 26% on building the shared VHT. It can be claimed that these queries could be accelerated when the VHT could be built in parallel.

Table 2.3.: Shared VHT build time (on 64 processors)

Query	Shared VHT build time in %
3	26.4
5	13.5
7	10.2
8	9.1
9	5.9
17	0.1
18	44.4
20	1.3
21	0.1

Parallel shared VHT building: The sequential building of the VHT is a scalability issue which forces multiple streams to be combined into one which then builds the VHT. Thus introducing synchronization overhead and giving away the possible performance improvement by multiple threads hashing and inserting into the VHT in parallel.

Independent *HashJoin* operators: The scalability of independent *HashJoin* operators is limited by the *XchgHashSplit* operators which is dicussed in Section 2.2.3.

2.3.4. *Reuse* operator

The *Reuse* materializes tuples to be reused in other parts of the QEP. In Vectorwise this operator is implemented completely sequential and is not parallelized in any way. This forces for example in Q2 the parallel streams to be merged and, after the reuse, to be partitioned again which introduces additional synchronization overhead for an operator which already consumes ≈ 19% of the whole query time.

Table 2.4 shows - for each query containing a *Reuse* operator - the percentage of query time spent in the *Reuse* operator and the percentage of (query) time spent in the operators consuming the reused tuples.

Parallel-Reuse: Having a *Reuse* operator being aware of parallelism would avoid that bottleneck and would eleminate the - in this case - unnecessary synchronization.

Table 2.4.: Time spent in Reuse (on 32 processors)

Query	Time spent in %
2	18.9
11	< 0.5
13	7.6
15	12.3
17	5.3
19	≈ 0
22	≈ 0

2.3.5. Locking

Locking protects a critical section against concurrent access so that only one thread can run intructions inside the critical section at a time. Thus this time spent in the critical section can be counted as sequential fraction of a parallel program.

In Vectorwise locking is used in many places e.g. locking a block in the buffer pool, during updates, ... where inside *Xchg* operators and in I/O layer are critical places that will be examined in the following.

Xchg

Each *Xchg* operator provides a synchronization point between producer and consumer threads where the shared state is protected against concurrent access using one mutex per *Xchg* operator. This mutex is acquired during setup of each producer or consumer thread, during query execution and during teardown. In each case the most time is spent on the search for a new buffer as explained in Section 2.2.2. Further that mutex is only acquired when a new buffer is needed or the producer or comsumer thread is woken up and the thread needs to check its sleeping condition again.

As known from Section 2.2.3 the *XchgHashSplit* operator is a victim of mutex contention.

Table 2.5 shows the time spent in the critical section t_{cont} of each TPC-H query where the critical section visualized is the one with the maximal t_{cont}. Further it shows that fraction of the whole parallelized query's time spent in this critical section. In Q10, Q13, Q14, Q16, Q19, Q20, Q21, Q22 this percentage reached to at least 10%.

Further a lock-free solution has been proposed in Section 2.2.5.

Table 2.5.: Contention time in *Xchg* mutex (on 64 processors)

Query	Contention time t_{cont}	Fraction of query time in %
1	$4.1 \cdot 10^6$	0
2	$2.3 \cdot 10^7$	0.5
3	$5.7 \cdot 10^7$	1.1
4	$1.5 \cdot 10^6$	0.1
5	$1.3 \cdot 10^7$	0.1
6	$2.3 \cdot 10^6$	0.2
7	$4.4 \cdot 10^6$	0.1
8	$6.6 \cdot 10^7$	0.4
9	$1.9 \cdot 10^8$	0.2
10	$4.3 \cdot 10^9$	10.1
11	$2.4 \cdot 10^6$	0
12	$3 \cdot 10^6$	0.1
13	$2.1 \cdot 10^{10}$	34.2
14	$1.3 \cdot 10^9$	13.3
15	$3.8 \cdot 10^8$	8.8
16	$1.7 \cdot 10^9$	11
17	$2 \cdot 10^8$	2.5
18	$7.7 \cdot 10^7$	0.2
19	$4.5 \cdot 10^9$	18.1
20	$1.5 \cdot 10^9$	11.2
21	$9 \cdot 10^9$	21.8
22	$1.6 \cdot 10^9$	15.1

PBM

It was noticed that during the initialization of the *MScan* operator there was a relatively high cost per (afterwards scanned) tuple. This was traced to the I/O layer of Vectorwise, which uses a component called PBM[4] [SBZ12]. The PBM manages buffer replacement of the buffer pool. It is based on the idea that the "buffer manager has quite a bit of information on the workload in the immediate future" [SBZ12, p. 1759] by actively watching "the progress of all scan queries" [SBZ12, p. 1759]. Based on this information it is possible to "to estimate the time of next consumption of each disk page" [SBZ12, p. 1759] and create "an approximation of the ideal [...] algorithm" [SBZ12, p. 1759]. This involves locking to protect PBM's workload metadata against concurrent access.

Figure 2.9 shows two TPC-H runs one with the standard configuration with PBM enabled and another one with the fallback LRU strategy. It can be seen that the PBM introduces noticeable overhead in Q9 and Q18.

Further after analyzing the profiling information gathered this overhead was caused by the

[4] Predictive buffer manager

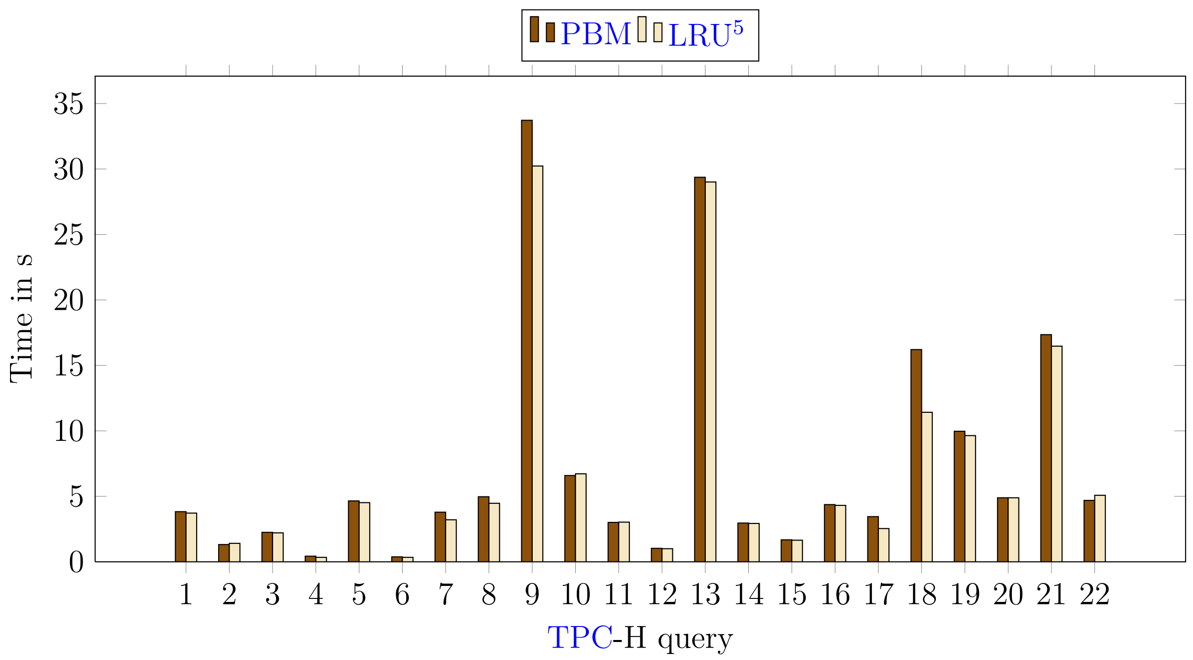


Figure 2.9.: Query response times with PBM and with the LRU strategy (on 64 processors)

locking inside PBM which was critical during startup and teardown of a *MScan* operator when many threads are scanning in parallel.

With the help of the profiling information, it is possible to get an overview about the overhead introduced by locking. This was done for the main mutex in PBM (*PBM main lock*). Table 2.6 visualizes the time the mutex was contented (contention time t_{cont}), its fraction of the query response time and the time spent waiting on the mutex (acquisition time t_{acq}). It can be seen that the fraction of query time spent inside PBM's main mutex is extreme for Q6 and Q17. But for most queries this fraction is under 5%. Consider Q9's contention time which is around $\frac{1.6}{2.3} = 0.7$s spent sequential. Note that a rather long contention time can imply an a relatively high acquisition time, which is 10^{10} cycles in case of Q9. From the t_{acq} it is possible to calculate the average time that each processors spent waiting, in Q9 this is $\frac{10^{10}}{64} = 1.5 \cdot 10^8$ cycles that are not spent on doing work.

2.3.6. Summary

In the previous sections it had been shown that in some queries - for example Q11 - there is an high ($> 90\%$) percentage of work done in sequential compared to the percentage (of work) done in parallel. Figure 2.10 visualizes the work that has to be done sequentially per query as a fraction of the query time where each query was parallelized. Note that this plot cannot show every part of the query that is processed sequentially, because

Table 2.6.: Contention and acquisition time in *PBM main lock* (on 64 processors)

Query	Contention time t_{cont}	Fraction of query time in %	Acquisition time t_{acq}
1	$4.5 \cdot 10^8$	3.1	$4.6 \cdot 10^9$
2	$2.1 \cdot 10^7$	0.5	$7{\cdot}10^6$
3	$2.3 \cdot 10^8$	4.3	$5{\cdot}10^9$
4	$7.2 \cdot 10^7$	6	$1.2 \cdot 10^9$
5	$2.9 \cdot 10^8$	2.7	$6.8 \cdot 10^9$
6	$1.1 \cdot 10^8$	8.1	$1.9 \cdot 10^9$
7	$2.5 \cdot 10^8$	3	$3.2 \cdot 10^9$
8	$2.8 \cdot 10^8$	1.8	$1.3 \cdot 10^9$
9	$1.6 \cdot 10^9$	2	$1{\cdot}10^{10}$
10	$2{\cdot}10^8$	0.5	$1.3 \cdot 10^9$
11	$3.7 \cdot 10^7$	0.1	$1.7 \cdot 10^8$
12	$1.4 \cdot 10^8$	4	$4.1 \cdot 10^9$
13	$1.4 \cdot 10^8$	0.2	$3.9 \cdot 10^8$
14	$3.3 \cdot 10^8$	3.4	$2{\cdot}10^9$
15	$1.6 \cdot 10^8$	3.8	$1.3 \cdot 10^9$
16	$3.5 \cdot 10^7$	0.2	$2.5 \cdot 10^8$
17	$9.8 \cdot 10^8$	12	$9.7 \cdot 10^9$
18	$3.9 \cdot 10^8$	1.2	$4.8 \cdot 10^9$
19	$3.4 \cdot 10^8$	1.4	$1.8 \cdot 10^9$
20	$3.1 \cdot 10^8$	2.3	$1.4 \cdot 10^9$
21	$4.9 \cdot 10^8$	1.2	$1.8 \cdot 10^9$
22	$1.5 \cdot 10^8$	1.4	$5.2 \cdot 10^8$

e.g. different mutexes may interfere (it is possible to lock a second mutex while being in the critical section of a mutex), Further effects like waiting times are not included in Figure 2.10. But still, Figure 2.10 can be seen as an per-query indication of scalability issues. This sequential fractions shown include:

- The - in Section 2.3.1 explained - *sequential phases* except for the query execution phase (*execution*).
- Furthermore it includes the time spent for *building shared VHT(s)* which are explained in Section 2.3.3 and
- parts of the QEP executed sequentially (*sequential parts* further explained in Section 2.3.2) and
- the length of the critical section of two mutexes: the PBM's main mutex and *Xchg* operator's mutex with the longest critical section - as explained in Section 2.3.5.

Furthermore it can be seen that most queries have a relatively high sequential fraction (in the parallelized case) where linear scalability by Amdahl's law would still require such a sequential fraction to be zero or at leat near zero.

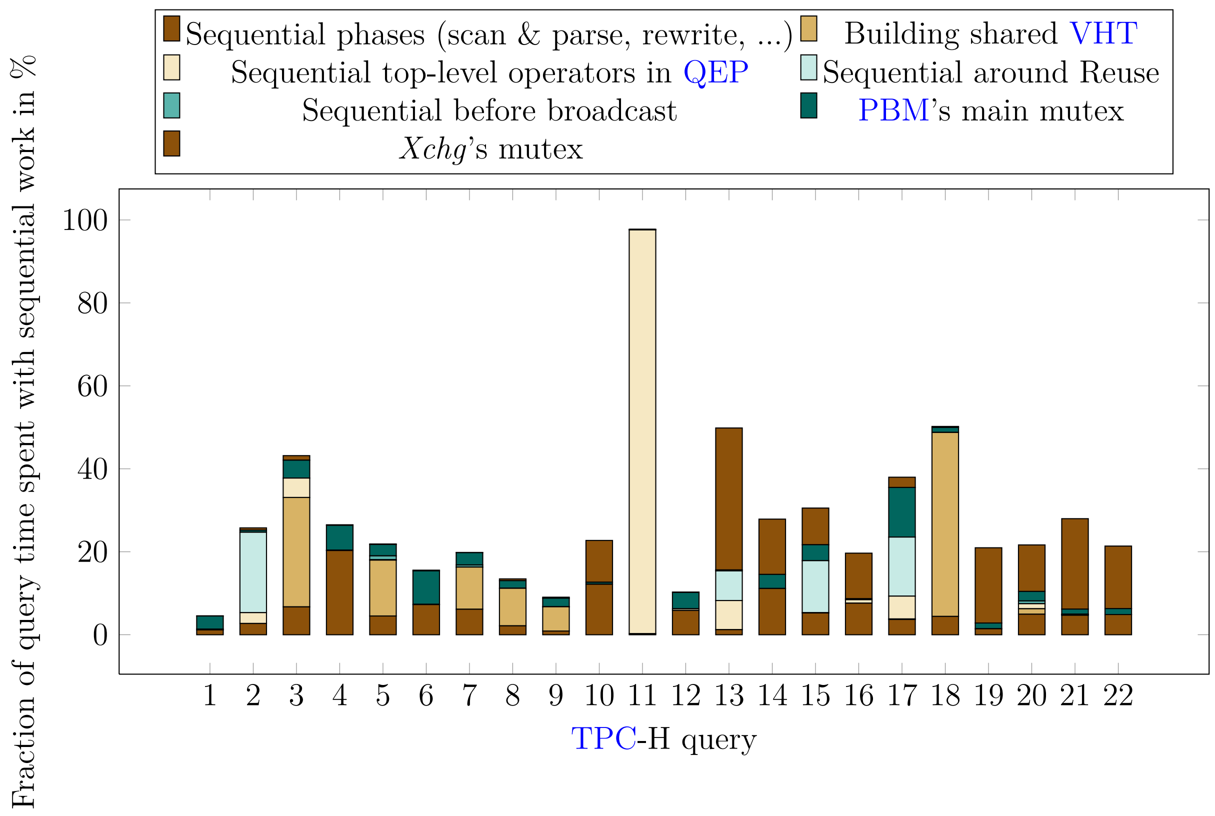


Figure 2.10.: Time spent sequential in % of query time (on 64 processors)

2.4. Memory locality

In order to find out the properties of Vectorwise's memory locality and in order to spot potential bottlenecks the following experiment was done: TPC-H was ran one time with no changes i.e. vanilla Vectorwise and one time with all memory of the *x100_server* process distributed round-robin over all NUMA nodes on a per-page granularity, which is referred to as *interleaved memory*. This involves a high probability of having to access memory on a remote NUMA node which will cause additional overheads, but reduces the probability of only a few NUMA nodes getting the bottleneck for memory access.

This led to the query response times as visualized in Figure 2.11. Further this led to an improved query response time, in the case of Q1, Q3, Q5, Q6, Q7, Q8, Q9, Q12, as well as regressed query response times in Q10, Q11, Q13, Q14, Q15, Q16, Q18, Q19, ..., Q22. It can be summarized that using interleaved memory has mixed influences on performance.

Furthermore interleaved memory had caused an extreme improvement in Q9, what almost halved the query response time.

In the following the memory locality of the *MScan* operator and the shared VHT *HashJoin* will be analyzed.

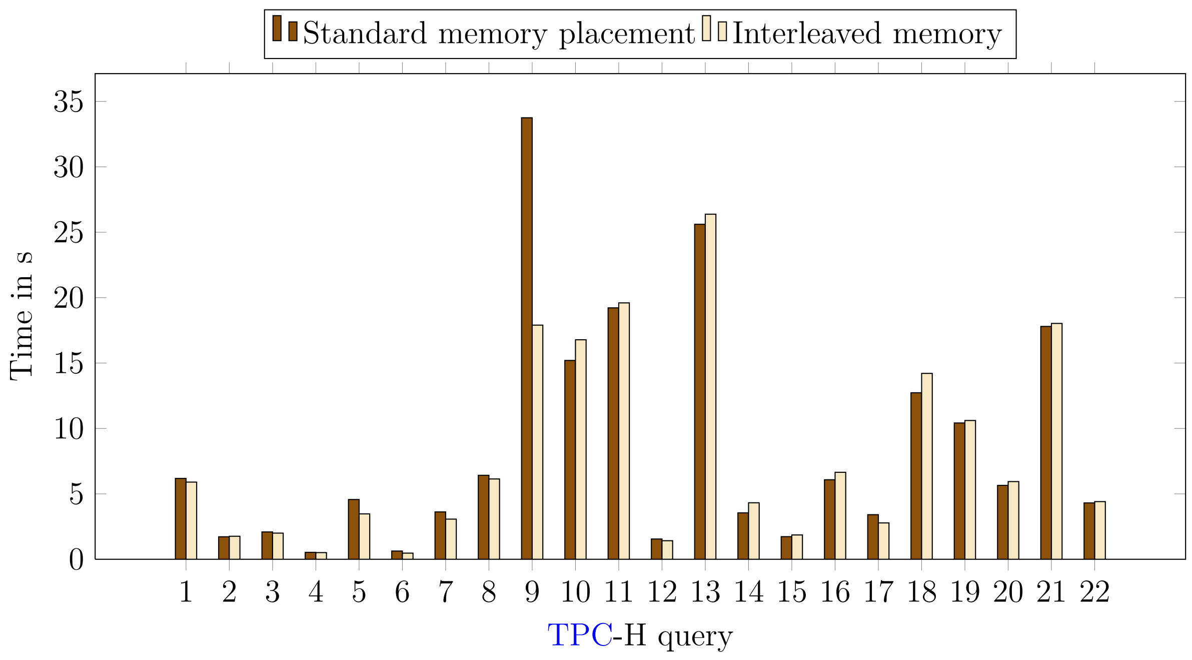


Figure 2.11.: Memory placement (on 32 processors)

2.4.1. *MScan* operator

In the tested scenario all *MScans* read the data from the blocks already cached in-memory.

The blocks were read by I/O worker threads which issue the read requests from persistent memory. These threads are not bound to any processor or NUMA node. That implies that even when Linux' first-touch policy can be applied the data of block might reside on a more or less random NUMA node. This implies that the reading/decompression speeds will be varying which is the case. As an example, in Q6 the columns *quantity*, *extendedprice*, *discount* and *shipdate* are scanned. Table 2.7 shows the times needed for decompression per column in $\frac{\text{cycles}}{\text{tuple}}$ over three runs. It can be seen that the minimum and the maximum differs by a factor of at least 2.

Table 2.7.: Decompression speed of Q6's scanned columns

Column	Minimum	Mean	Maximum
discount	3.1	6.9	15.3
extendedprice	3.7	8.8	15.9
quantity	5.2	9.5	16.7
shipdate	4.1	7.3	14.4

An experiment was performed in which Q6 was ran without changes and with interleaved memory, which forces all (memory) pages to be distributed over all NUMA nodes in a round-robin fashion. Table 2.8 shows the decompression time which is comparable to Table 2.7. It can be seen that the minimum speed is approximatetly equal (the difference

Table 2.8.: Decompression speed of Q6's scanned columns with interleaved memory

Column	Minimum	Mean	Maximum
discount	2.7	3.2	6.4
extendedprice	3.4	3.9	7.1
quantity	4.6	5.5	11.5
shipdate	3.9	4.8	10

is $< 1 \frac{\text{cycles}}{\text{tuple}}$).

Further it can be seen that the mean, represented by the column *Mean*, approximately halved. In other words the variant with "no" locality (i.e. interleaved memory, very low probability for local access) is - in average - faster than the standard strategy.

This leads to the conclusions that:

1. the standard strategy is far from optimal (in Q6) and that
2. it must be limited by the memory bandwidth, which is caused by a few nodes - on which the blocks reside - becoming the bottleneck for memory access.

Note that the second conclusion can also be drawn from the maximal time of the decompression (column *Maximum*) which reduces by $\geq 5 \frac{\text{cycles}}{\text{tuple}}$.

2.4.2. Shared VHT *HashJoin*

The - in Section 2.3.3 - explained parallelized *HashJoin* where one thread builds a VHT and all parallel threads probe through this shared VHT afterwards.

One problem is that the one thread that builds the VHT makes its memory region local to its NUMA node (first-touch policy), because it is the first who writes to it. That implies that in a scenario using $n \geq 8$ (kernel) threads the VHT is local to only 8 threads and all other $n - 8$ threads have to access the VHT through remote memory access. In a worst-case scenario the VHT will be accessed remotely by all threads.

This leads to slower access in sense of latency as described in Section 1.2. Further the NUMA node hosting the VHT may become a bottleneck.

Table 2.9.: Shared VHT *HashJoin* probing time in $\frac{\text{cycles}}{\text{input tuple}}$ in Q3

Memory	Minimum	Mean	Maximum
standard	304.39	727.74	1,881.59
interleaved	194.64	243.4	334.87

Table 2.9 shows the time needed probing one tuple in the VHT in $\frac{\text{cycles}}{\text{input tuples}}$, with forced

interleaved memory and with the standard memory allocation that Vectorwise DBMS provides. More precisely it shows the shared VHT *HashJoin* in Q3 in which 31 threads are probing concurrently through the VHT . The thread which builds the VHT and after the build probes was excluded.

Using interleaved memory decreased the per-tuple cost in general. This leads to the conclusion that using interleaved memory reduces single NUMA node / or only a few NUMA nodes being a bottleneck for memory access in parallel *HashJoin* operators using a shared VHT.

2.4.3. Summary

In Section 2.4.1 and Section 2.4.2 it was shown that the query response time could be reduced by using interleaved memory when only a few NUMA nodes are becoming the bottleneck for memory access. In case of the shared VHT *HashJoin* in Section 2.4.2 the thread building the VHT sets the memory locality to its local NUMA node which makes this NUMA node a possible candidate for being a bottleneck for memory access.

Proposed solution: The bandwidth limitation due to the single NUMA node becoming a bottleneck can be overcome by using interleaved memory for the buffer pool and for the shared VHT. This allows to balance the memory accesses over all NUMA nodes and hence eleminating the single node bottleneck as such.

2.5. Skew

According to Amdahl's law, the maximal speedup can be reached when the sequential percentage is zero and all parallel streams are running for an equally long time. But in practice this is almost never the case. Typically these different parallel stream have different processing speeds. This might be due to memory locality (e.g. cold caches, NUMA), OS[6] scheduling issues (e.g. different time quantums), etc.

Consider an example where one query is processed which consists of two pipelines e.g. first building a hash table and afterwards probing tuples through the built hash table. Each pipeline is executed using 4 kernel threads which run parallel to each other. It is highly probable that not all threads complete their work at the same time. Consider the drawing in Figure 2.12 which visualizes this example. Note all threads of the first

[6] Operating system

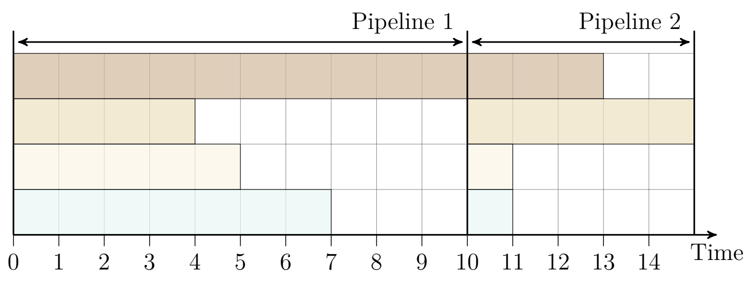


Figure 2.12.: Skew in a query consisting of two pipelines

pipeline (*Pipeline 1*) need to be completed, before the second pipeline (*Pipeline 2*) can be evaluated.

Vectorwise maximally uses as much threads as available processors, which implies that given different processing speeds, threads complete their work later or earlier than the "average" thread. That makes linear scalability impossible, because it requires full utilization of the processors which cannot be reached anymore.

This *imbalance* is referred as *execution skew*. Further *execution skew* could also occur in the presence of imbalanced data distributions. This is referred as *data skew*. There are multiple ways *data skew* can be introduced. Two are described in the following:

- on the hand-side the data may not be partitioned into equal-sized chunks
- or the parallel running QEP operators may have different selectivities and therefore discard different amounts of tuples.

Effects: Further the maximal speedup gained in the presence of skew was evaluated in [BFV99, p. 106] using a model, based on the following assumptions:

- According to [BFV99, p. 106] that model assumes that only one operator introduces skew.
- That, the one operator introducing skew, has a rank r which determines its position in the pipeline chain where $r = 1$ determines the top-most operator in the pipeline, as stated by [BFV99, p. 106].
- Further - according to [BFV99, p. 106] - it was assumed that every operator has the same cost.
- Skew introduces using a "tuple [that] produces k % of that operator's result" [BFV99, p. 106] where "p threads allocated to the query" [BFV99, p. 106].

Figure 2.13 shows the maximal speedup reached by different fractions of skew k for different positions r of the operator that introduces skew. Further it can be seen that the

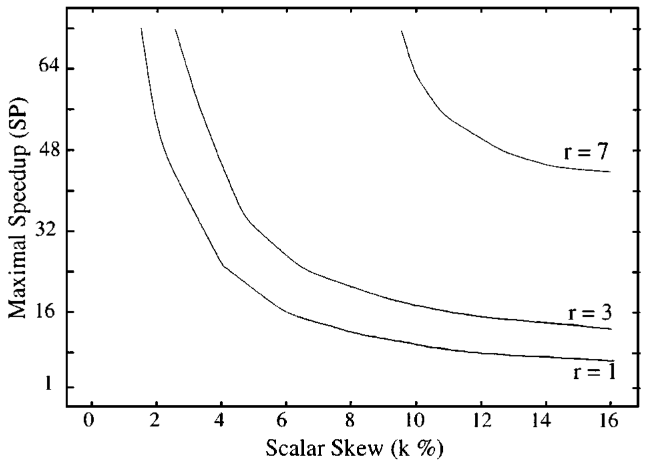


Figure 2.13.: Maximal speedup in the presence of skew [BFV99, p. 106]

”speed-up gets worse as the skewed operator reaches the beginning of the chain” [BFV99, p. 106] and that the maximal speedup reached it limited by the ”skew factor k” [BFV99, p. 106] and, obviously, ”the number of threads p” [BFV99, p. 106].

Existence in queries: The execution skew S over a set of thread runtimes T can be quantified by the difference between the maximal runtime and the minimal runtime of a thread which leads to

$$S = \frac{max(T) - min(T)}{max(T)}$$

under the assumption that each thread started at the same time.

Table 2.10.: Skew S below the top-most *Xchg* operator over TPC-H query set restricted by queries with only one *Xchg* operator

Query	Skew $S \cdot 100$
Q1	31
Q4	66
Q6	29
Q12	38

This formula leads for a subset of the TPC-H queries (with one *Xchg* operator only) to results as can be found in Table 2.10. In these queries it was found out that not all thread process an equal amount of tuples (over their whole QEP). Further if threads process a nearly equal amount the processing speeds of each thread differs which is caused e.g. by different memory access times.

3. Related work

This chapter will present recent research done about general parallelism models, user-level scheduling and progress estimation. Further it will be anaylzed whether these approaches are usable in Vectorwise, regarding parallelism models and in the frame of this thesis.

3.1. Parallelism model

Because of the disadvantages of the Volcano-model parallelism, the recently developed database systems HyPer and BLU chose parallelism models that drifted away from the *exchange*-base parallelism towards task-based parallelism.

Morsel-driven parallelism: [LBKN14] proposes a concept in which the data is partitioned into fine grained chunks, called *morsels*. These morsels could run in parallel to each other and are dynamically distributed over the available cores by a *Dispatcher*. This allows to "to assign a core to a different query at any time" [LBKN14, p. 5] which further "also guarantees load balancing and skew resistance" [LBKN14, p. 5]. In general all operators are aware of parallelism and provide implementations optimized for NUMA locality and parallelism. For example [LBKN14] proposes a lock-free and parallel implementation of the Hash-Join algorithm with the hash table being aware of NUMA.

Note that in Hyper was written from scratch with operators being aware of parallelism which is not easily feasible in Vectorwise, because the parallelism model implemented (Volcano-model parallelism) keeps operators unaware of parallelism. In this thesis we try to see how principles from such system can be used to refloat the Volcano model, using in my existing database systems.

BLU-like parallelism: As stated in [RAB+13, p. 1083], BLU divides a query into potentially multiple, so called, *single-table queries*. Each for these will be executed using a set of kernel threads. Further this model was implemented from scratch with operators being aware of parallelism which makes it hard to be integrated into the existing Vectorwise code base.

3.2. User-level scheduler

The drawbacks of heavyweight, kernel-supported threading such as pthreads are well-known [...], leading to the development of a plethora of user-level threading models" [WMT08, p. 7]: Most approaches (e.g. Intel TBB[1] ([Rei07, p. 2]), even Hyper ([LBKN14, p. 4])) concentrate on task-level scheduling which decomposes bigger parts of work into possibly many tasks and tries to keep all cores/processors busy with executing tasks (through using a pool of worker (kernel) threads). The schedulers in TBB and Hyper do not support context switching in the sense of switching from one task (A) to another (B) and afterwards continuing with A. But this makes them hardly usable in Vectorwise where thread can wait a long period on a resource to be available. In the case of Intel TBB this would block a worker thread, or a morsel in case of morsel-driven parallelism ([LBKN14]). Some of these waits occur in the implementation of the Volcano-model alike *Exchange* operator and in the implementation of the scan operators where a scan would wait on I/O[2]. Both would require many distributed changes in different components to fit longer waits into the task-driven parallelism model.

It is possible to avoid these changes with user-level scheduling that allows context-switching inside a unit of work (a user-level thread).

Protothreads are "extremely lightweight stackless [...] threads" [DS05, p. 6] which "only requires only two bytes of memory per protothread" [Dun14]. As stated by [DS05, p. 6], Protothread base on the concept of, so called, *local continuations*. In the frame of [DS05] these local continuations are implemented using a function f, a state s and the C *switch* statement on the state s in combination with a *case* statement for each state. "A protothread is driven by repeated calls to the function [f] in which the protothread is running" [Dun14] in combination with a starting state $s = 0$. Then via the *switch* statement the control flow jumps to the case where $s = 0$ is handled. In general this allows it to jump to any state s with a *case* statement. In case of a context switch between different Protothreads, the s is set to the next state for the Protothread which shall be left, allowing to continue at the new state. After setting the next state the *return* statement is used to leave the function f which allows another function to run.

This concept of threads comes with the advantage of being lightweigth, because the stacks can be shared between the Protothreads implying that very many of these can be run without excessive memory usage. The cost of this advantage is that "automatic variables - variables with function-local scope that are automatically allocated on the stack - are not

[1]Thread Building Blocks
[2]Input / output

saved across a blocking wait" [DS05, p. 9] where "blocking wait" is equivalent to a context switch. This disadvantage makes Protothreads - despite its advantage of being lightweight - a bad fit, because it cannot be guaranteed that these automatic variables are saved between context-switches. Allowing to save these automic variable across context-switches will require the usage of stack memory and explecitely saving the current context of execution as it is done by the following.

Shared-stack Cooperative Threads try to reduce the memory overhead that is involved with many threads using shared-stacks as stated in [GKHC07, p. 1184] where each "cooperative thread occupies a fixed-size stack like the way of multithreaded systems" [GKHC07, p. 1184].

[GKHC07] works around having a fixed-sized stack for each thread by differentating between the stack of active threads and inactive threads. "There is a single shared-stack used as a runtime stack for all threads [...] which is called the shared-stack" [GKHC07, p. 1184] where "only the stack of the currently running thread occupies the shared-stack". At a context-switch "it allocates a buffer in heap and copies its thread context to the buffer" [GKHC07, p. 1184] whereas the "thread that will run next copies its thread context to the shared-stack and resumes its execution" [GKHC07, p. 1184].

GNU Pth allows to emulate multiple threads. Further "Pth doesn't require any kernel support, but can NOT benefit from multiprocessor machines" [Eng14] where that restriction is a major one, because it explicitely excludes the use of multiple processors.

Qthreads "provides basic lightweight thread control and synchronization primitives in a way that is portable to existing highly parallel architectures" [WMT08, p. 7]. "This technique marks each word in memory with a "full" or "empty" state, allows programs to wait for either state, and makes the state change atomically with the words contents" [WMT08, p. 2].

3.3. Query progress estimation

"The need for accurate SQL progress estimation in the context of decision support administration has led to a number of techniques proposed for this task" [KDCN11, p. 382]:

Machine learning: [MZZ08] proposes a concept of learning how long a query takes with a given system state. The authors propose a machine learning technique of creating

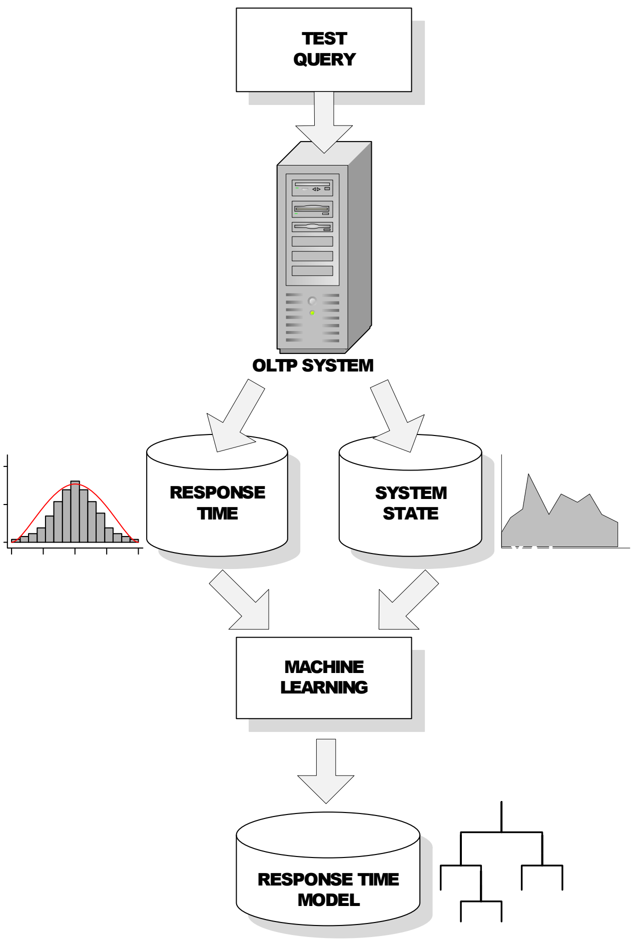


Figure 3.1.: Progress estimation by machine learning [MZZ08, p. 491]

a classification tree based on the system's state in combination with the query during the learning phase as visualized in Figure 3.1. After the learning phase the created classification model is used to estimate query's progress.

***GetNext()* model:** [CNR04] describes an alternative way to determine to progress of a QEP based on the number of *GetNext()* calls of an operator in the QEP in combination with the type of operator (e.g. *Scan*, *Join*, ...). That method "assumes that the total work (i.e., CPU overhead, I/O, etc.) is amortized across the *GetNext* calls issued across all nodes in the execution plan. Hence, the fraction of the total *GetNext* calls executed at any point in a query can be used as an estimate of its progress" [KDCN11, p. 384] where total *GetNext* calls means the "total number of *GetNext()* calls that will be performed over all nodes in the query" [CNR04, p. 806]. [CNR04] states that estimating the total "number of *GetNext()* calls for an operator in the query execution plan is the cardinality estimation problem faced by query optimizers" [CNR04, p. 805].

As stated by [CNR04] the progress of a single pipeline can be estimated by estimating its - so called - *Driver Node* which is the node/operator that introduces data into the pipeline (e.g. *Scan* operators, *Sort* operators when they output their sorted tuples ...). Further [CNR04] defines the *Driver Node*'s estimated progress the fraction of the number of *GetNext* calls in the *Driver Node* divided by the estimated total number of *GetNext* calls in the *Driver Node*.

In [CNR04] the authors extend the scheme for queries consisting of multiple pipelines by summing, per pipeline, the current number of *GetNext()* calls divided by the sum of all estimated total number of *GetNext* calls of each pipeline. In [CNR04] each pipeline is represented using its *Driver Node* while utilizing optimizer's cardinality estimates for deriving the total number of *GetNext* calls.

The in [CNR04] presented model was refined by [MK07] and [KDCN11].

4. Progress estimation

The technique - presented in this chapter - shall estimate the progress of a given QEP and will be used in Chapter 5 to estimate the time left until a stage of query execution is completed and as criterion of priority between QEPs in the same scenario as explained in Chapter 2. As it will be used for assessing the relative progress between threads running the same QEP and not the overall QEP progress, it differs from query progress estimation techniques like these presented in Section 3.3.

Desirable properties: According to [MZZ08, p. 490] desirable properties for query progress estimation are:

- Monotonicity i.e. the reported progresses cannot go ”backwards”,
- Low overhead, low interference with query execution and
- ”Leveraging feedback from execution: as query execution progresses, more information based on (intermediate) results of execution can become available. Ideally, an estimator should be able to take full advantage of such information” [MZZ08, p. 490].

Simplifications: Given the later usage of the progress estimation scheme and the scenario the following simplifications can be made:

- I/O does not need to be considered.
- Disk spilling is not considered.
- Scan ranges are static.
- Futhermore for the usage inside the user-level scheduler the progress estimation needs to be fine granular estimation inside a stage of query execution where other estimation techniques - as described in Section 3.3 - want a high accuracy for the estimation of whole query, which is not needed.

4.1. Approach

Consider an example query like described in Figure 4.1 which consists of two physical operators:

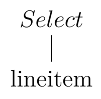


Figure 4.1.: Simple example QEP

- A *Scan* which reads the relation *lineitem* and
- a *Select* operator that filters out tuples which do not match a given predicate.

The progress of this query could be described by the progress of the *Scan* operator which could be measured by the amount of tuples already read ($produced$) and the total number of tuples to read $total$. This implies that such a progress $progress(Scan)$ can be measured by $progress(Scan) = \frac{produced}{total}$.

Further to construct the progress of the QEP the progress of the *Select* operator has to be measured. There are different ways with different advantages and disadvantages in order to achieve that.

- It is possible to describe the progress of the *Select* operator analogue to the progress of the *Scan* operator. In that case the *Select* operator needs to know the cardinality at its output which then could be used as $total$. This cardinality could be gathered by estimates from the optimizer. These estimates are created on the basis of statistics gathered by the system and may be extremely inaccurate due to potentially coarse grained statistics.
- Further it is possible to ignore the progress of the *Select* operator by using the progress of the *Scan* operator. This implies that only the progress of the operator introducing data into the QEP is measured. In the context of this thesis this is referred to as *Source progress*. On one hand-side this completely avoids using the optimizer's cardinality estimates. But on the other hand this may not exactly represent the real progress at the *Select* operator. Consider the following example: Given that the progress of the *Scan* operator is 1 or - in other words - the *Scan* operator read everything it is supposed to read. By applying the previously mentioned concept the progress of the *Select* operator would be 1 too, even if there is a stride left to process. This happens after the stride has been produced by the *Scan* operator and not yet consumed by the *Select* operator.

Summarizing this example it is possible to estimate the progress by estimating the progress of the operator which introduces data into the QEP.

Aggr: This scheme can be extended for more complex QEPs. Consider a query defined by the QEP in Figure 4.2, which e.g. eleminates duplicates of the previous example query (Figure 4.1) via the *Aggr* operator.

$$\begin{tikzcd}[row sep=small] \mathit{Aggr} \arrow[d, dash] \\ \mathit{Select} \arrow[d, dash] \\ \text{lineitem} \end{tikzcd}$$

Figure 4.2.: More complex example QEP

The *Aggr* operator has two different phases. In the first phase it consumes all input tuples and stores it. Afterwards it reads the stored tuples in order to produce its resulting tuples, which is the second phase.

The progress of the first phase can be estimated by estimating the operator that introduces data into the QEP i.e. the operator scanning *lineitem*. This progress may not necessarily represent the progress of the whole *Aggr* operator which happens e.g. when time spent in the operators above in the QEP is large.

Further the progress of the second phase can be estimated by seeing the second phase as a virtual *Scan* operator on the stored values. Also this progress may not necessarily represent the progress of the whole *Aggr* operator, e.g. it may happen that the time spent in the sub-tree below the *Aggr* operator is significantly larger than the time spent in the operators above.

As can be seen none of both estimated progresses of the first phase and the second phase estimates the progress accurately in all cases. But the estimated progress could be made a tuple or vector instead of a scalar value where one element defines in which phase the QEP is in and another element that represents the estimated progress of the current phase.

4.2. Definition

Defining the estimated progress of an QEP as a tuple $[stage, advancement] \in Progress$ where

$$Progress := \Big\{[stage, advancement] \in (\mathbb{N} \cup \{0\}) \times \mathbb{Q} \;\Big|\; (0 \leq advancement \leq 1)\Big\}.$$

Stage: It is possible to partition query execution into different consecutive phases which are run sequentially. The *stage* of the $[stage, advancement]$ tuple describes in which phase the query execution is in and is further called stage number. Consider the example QEP visualized in Figure 4.3. Visualized is a QEP consisting the following operators: *Aggr*,

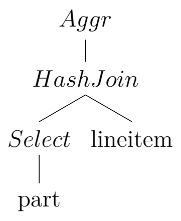


Figure 4.3.: Example QEP

HashJoin, *Select* and the scans of the relations *part* and *lineitem*. The *Aggr* and *HashJoin* operators have blocking phases. Implying that the whole QEP (Figure 4.3) has in sum 3 phases which are executed sequentially

- At first the phase called *HashJoin build* is executed. This phase involves scanning the relation *part*, selecting tuples by the *Select* operator and putting them in the hash table of the *HashJoin* operator.
- The second phase *HashJoin probe & Aggr build* runs after the first phase (*HashJoin build*) has been completed and involves scanning the relation *lineitem*, probing them through the hash table of the *HashJoin* operator and materializing them (the tuples for which exists a match in *HashJoin*'s hash table i.e. the not discarded tuples) into the hash table of the *Aggr* operator.
- The last phase of the example QEP is called *Aggr produce*. It involves producing the result tuples from the *Aggr*'s hash table.

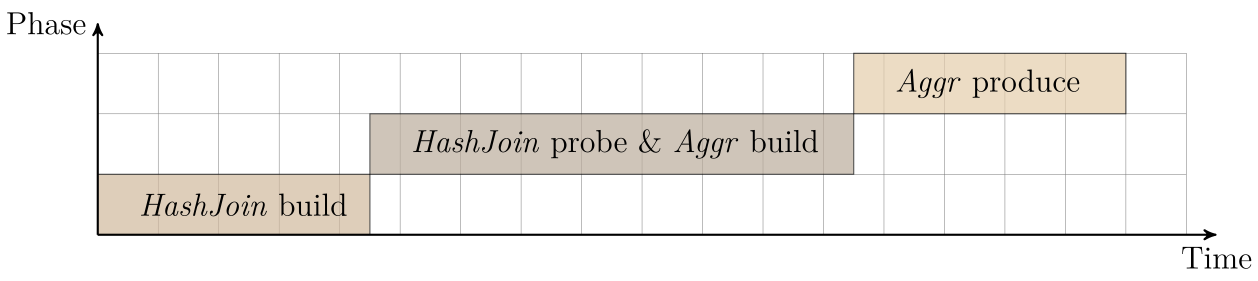


Figure 4.4.: Phases of the example QEP

Figure 4.4 shows the sequential execution of these phases (*HashJoin build*, *HashJoin probe & Aggr build* and *Aggr produce*). Claiming that it is possible the estimate the progress of a (sub-)QEP inside such a phase, it is also possible to estimate the progress of the whole QEP by giving each phase a stage number. The stage number given to each phase is determined by the sequential execution order starting with stage number 0. In the example (see Figure 4.3 and Figure 4.4) the 3 phases will get the following stage numbers:

- *HashJoin build* will get the stage number 0 because it is the first phase,
- the second phase (*HashJoin probe & Aggr build*) will get the number 1 and
- the third/last phase called *Aggr produce* will the stage number 2.

A stage number has the following properties:

- The stage number $stage$ is bounded by 0 and the maximal stage number $stage_{max}(op)$ of a QEP with its top-most operator $op \in Operator$ i.e. $0 \leq stage \leq stage_{max}(op)$ Note that the maximal stage number can be determined by counting the blocking phases and adding 1.
- It (the stage number) is strictly monotonic increasing and
- contains only consecutive numbers.

Advancement: In the previous paragraph it was claimed that it is possible to estimate the progress inside a phase of query execution. This (progress inside a phase) is described by *advancement* of the $[stage, advancement]$ tuple.

Each phase of query execution reads a determinable number of tuples. For example: A *Scan* operator "knows" how many tuples it will scan and which tuple is it currently processing. For a *Aggr* operator it is possible to determine the number of tuples materialized in the hash table and the current tuple at the output of the *Aggr* operator.

Through this fact it is possible to express the progress inside a phase and combined with the stage number (i.e. $[stage, advancement]$) it is possible to estimate the progress of a whole QEP consisting of an abitrary number of phases as will be shown in the following.

Progress estimation function: In order to retrive the estimated progress of a given QEP, represented by its top-most operator $op \in Operator$, it is necessary to have a function $progress(op) : Operator \to Progress$. This function calculates a progress $p \in Progress$ from the state of the operator op.

4.3. Conventions

In the following general conventions will be defined in order to make the definition of estimated progress short and concise.

- nil determines a special value and is defined as $[0, 0]$
- $left$ and $right$ denote the children of a operator. With being $left$ the left child and $right$ the right child. In the case of unary operators (for example $Aggr$) both ($left$ and $right$) represent the same child.
- Based on the defined progress the relations $<$, $>$, $=$, $\leq$ and $\geq$, with $S \subset Progress$; $a, b \in Progress$, can be defined :

$$\begin{aligned} a < b := & (stage(a) < stage(b)) \vee \\ & \big((stage(a) = stage(b)) \wedge (advancement(a) < advancement(b))\big) \end{aligned} \tag{4.1}$$

$$\begin{aligned} a > b := & (stage(a) > stage(b)) \vee \\ & \big((stage(a) = stage(b)) \wedge (advancement(a) > advancement(b))\big) \end{aligned} \tag{4.2}$$

$$a = b := (stage(a) = stage(b)) \ \wedge (advancement(a) = advancement(b)) \tag{4.3}$$

$$a \leq b := ((a < b) \vee (a = b)) \tag{4.4}$$

$$a \geq b := ((a > b) \vee (a = b)) \tag{4.5}$$

Based on the relations defined by Equation (4.1), Equation (4.1), Equation (4.3), Equation (4.4) and Equation (4.5) the minimum in a set of progresses S can be defined by Equation (4.6).

$$min(S) := \begin{cases} x \text{ where } \exists x \in S \setminus \{nil\} : \forall y \in S \setminus \{nil\} : x \leq y & \text{if } |S \setminus \{nil\}| \geq 2 \\ x \text{ where } x \in S & \text{if } |S \setminus \{nil\}| = 1 \\ nil & \text{otherwise} \end{cases} \tag{4.6}$$

Further analogue to the minimum it is possible to define the maximum over such a set S as defined by Equation (4.7).

$$max(S) := \begin{cases} x \text{ where } \exists x \in S : \forall y \in S : x \geq y & \text{if } |S \setminus \{nil\}| \geq 2 \\ x \text{ where } x \in S & \text{if } |S \setminus \{nil\}| = 1 \\ nil & \text{otherwise} \end{cases} \tag{4.7}$$

- Further it is helpful to have functions that provide direct access to the components of a progress $p \in Progress$ where $p = [stage', advancement']$:

$$stage : Progress \to \mathbb{N} \cup \{0\}$$
$$stage([stage', advancement']) := stage'$$

$$advancement : Progress \to \mathbb{Q}$$
$$advancement([stage', advancement']) := advancement'$$

- Each operator can have an annotation in the subscript of their name. These annotation describes the phase they are in or at which side (producer or consumer) they are. Phases are described by a G and possible phases are $G = build$, $G = probe$ and $G = produce$. Further sides are described by a *side* in the subscript, where possible sides are $side = Consumer$ and $side = Producer$.

Further the whole set of operators in Vectorwise DBMS can be partitioned into 4 sets of operators:

- *Scan* operators,
- *Streaming* operators,
- *Blocking* operators and
- *Buffering* operators.

4.4. Scan operators

Scan operators read data from a table and it is possible to specify ranges (of tuples) to be read.

Scan ranges: Based on that it is possible to define a scan range to be a tuple $[low_{RID}, high_{RID}]$ where low_{RID} and $high_{RID}$ defines an interval starting with the record at position low_{RID} and ends with $high_{RID}$ with $low_{RID}, high_{RID} \in \mathbb{N} \cup \{0\}$ and $low_{RID} \leq high_{RID}$. Note that a record represents a tuple. That imples that a scan range $[low_{RID}, high_{RID}]$ consists of $high_{RID} - low_{RID}$ tuples.

Furthermore a *Scan* operator is not restricted to only read tuples of one scan range, it reads a set of not overlapping scan ranges, called $ScanRanges$.

Progress: With the help of *ScanRanges* it is possible to determine the total number of tuples that a *Scan* operator will read. This total number is definied by the following equation:

$$tuples_{total} = \sum_{[low,high] \in ScanRanges} (high - low)$$

This total number of tuples $tuples_{total}$ will be only calculated once because the set of scan ranges ($ScanRanges$) is static for *Scan* operators and will therefore not change after the first call of the $next()$ function. In addition to that each *Scan* operator keeps track of how many tuples it has produced in sum which is denoted as $tuples_{produced}$. When combining both ($tuples_{produced}$ and $tuples_{total}$) it is possible to describe the progress of a *Scan* like defined in Table 4.1.

Note that in Vectorwise there is also an *Array* operator which generates data using various dimensions ($Dimensions$) where each dimension $d \in Dimensions$ has a size $size(d)$ and produces tuples from 0 to $size(d) - 1$. The *Array* operator produces the cartesian product of all these dimensions.

Table 4.1.: Scan operators

$$\begin{aligned} progress(MScan) &= [0, \frac{tuples_{produced}}{tuples_{total}}] \\ progress(Array) &= [0, \frac{tuples_{produced}}{\prod_{d \in Dimensions} size(d)}] \end{aligned}$$

This leads to the formulas that can be seen in Table 4.1 which defines the function $progress(op)$ for the operators *MScan* and *Array*.

4.5. Streaming operators

Streaming operators (in Vectorwise DBMS) only materialize a small amount of tuples in memory. Therefore it is assumed that these operators do not have a big influence on the progress. This leads to the progress formulas visualized in Table 4.2 where the progress of a *streaming* operator is represented by the progress of its child operator(s)/sub-tree(s).

Table 4.2.: Streaming operators

$$\begin{aligned} progress(MergeJoin) &= max\{progress(left), progress(right)\} \\ progress(OrdAggr) &= progress(left) \\ progress(Project) &= progress(left) \\ progress(Select) &= progress(left) \end{aligned}$$

4.6. Blocking operators

Blocking operators consume at least one input side fully.

Stages: *Blocking* operators have different stages in which they consume, process or produce tuples. These stages have a determined order which depends on the type of operator and are denoted using a G in the subscript of the operator.

In the following the different operators and their stages will be described:

Aggr: The *Aggr* operator has two stages:

- In the first stage it consumes all its child operator's tuples and stores them. This stage is referred as *build* stage and *Aggr* operator behaves like a streaming operator in this stage. Thus it uses the progress of its sub-tree.
- Afterwards *Aggr*'s second stage starts where it iterates of its stored tuples. This stage is called *produce* stage. In this stage the *Aggr* operator. Hence its introduces tuples it can be seen as a *Scan* operator over the virtual relation defined by the stored tuples over which is iterated.

This leads to the formulas for the *Aggr* operator as in Table 4.3.

HashJoin: The *HashJoin* operator also has two stages:

- During the *build* stage it consumes all tuples from its *right* child operator and inserts every tuples into a hash table. Hence - progress-wise - it can be seen as a *streaming* operator where the progress of its *right* sub-tree determines the progress of *HashJoin*'s *build* stage.
- After that stage, the *probe* stage follows, in which it consumes the tuples from its *left* child operator and tries to find a match in the hash table. If that is not the case, the tuple is discarded. Also in this stage - progress-wise - the *HashJoin* operator can be seen as a *streaming* operator where the progress of the *HashJoin* operator is determined by the progress of its *left* sub-tree.

Both stages lead to the formulas for the *HashJoin* operator as in Table 4.3.

AntiJoin/SemiJoin: Also other join algorithms like these implemented by the *AntiJoin* or *SemiJoin* operator have two stages like the *HashJoin* operator. Furthermore progress-wise they use the same formula as the *HashJoin* operator.

RevJoin/RevSemiJoin/RevAntiJoin: In Vectorwise there exists a class of operators implementing reverse joins. Compared to the *HashJoin* operator they have a third stage where - like in the *Aggr* - the result is produced through iterating over the VHT, leading to the formulas in Table 4.3.

Table 4.3.: Blocking operators

$$
\begin{aligned}
progress(Aggr_{G=build}) &= progress(left)\\
progress(Aggr_{G=produce}) &= \Big[stage(progress(left)) + 1, \tfrac{tuples_{produced}}{tuples_{in\ hash\ table}}\Big]\\
progress(Sort_{G=build}) &= progress(left)\\
progress(Sort_{G=produce}) &= \Big[stage(progress(left)) + 1, \tfrac{tuples_{produced}}{tuples_{materialized}}\Big]\\
progress(HashJoin_{G=build}) &= progress(right)\\
progress(HashJoin_{G=probe}) &= \Big[stage(max\{progress(left), progress(right)\}) + 1,\\
&\quad advancement(progress(left))\Big]\\
progress(HashRevJoin_{G=build}) &= progress(left)\\
progress(HashRevJoin_{G=probe}) &= \Big[stage(max\{progress(left), progress(right)\}) + 1,\\
&\quad advancement(progress(right)\Big]\\
progress(HashRevJoin_{G=produce}) &= \Big[stage(max\{progress(left), progress(right)\}) + 2,\\
&\quad \tfrac{tuples_{produced}}{tuples_{in\ hash\ table}}\Big]
\end{aligned}
$$

4.7. Buffering operators

Buffering operators can be seen as a hybrid of *streaming* operators and *blocking* operators, because they buffer an amount of tuples for a time span for later consumption by their respective parent operators.

Sides: *Buffering* operators have two sides:

- a *Produce* side which inserts tuples into a buffer and
- a *Consume* counterpart which reads tuples from a buffer.

The - in this document presented - formulation uses a subscript $op_{side=s}$ for a buffering operator $op \in Operator_{buffering} \subset Operator$ to signal which side $s \in Produce, Consume$ it represents.

Reuse: The *Reuse* operator allows to reuse tuples that are produces in a different subtree of the same query. According to this fact the progress of the consuming side of the *Reuse* operator ($Reuse_{side=conume}$) is the progress of producing side ($Reuse_{side=produce}$) at any time. This leads to the formulas for the *Reuse* operator as in Table 4.4.

Xchg: *Xchg* operators a operators which introduce parallelismn into a QEP. They create a set of threads for their *Produce* side. Each thread then retrieve the tuples from their child operators and stores these tuples into a shared (by all threads of one *Xchg* operator) set of buffers (called $Buffers$). These two sides are represented in the progress estimation model: Producer threads store the estimated progress of their child operators in the buffers. Consumer threads read (estimated) progress stored in the buffers.

Furthermore *Xchg* operators can have buffers for a specific consumer (for example *XchgHash-Split*). Therefore it is necessary to choose the "right" buffers for determining the progress of the child operator. This is modeled by the $select_{xchg}(buffer) : Buffers \to \{0,1\}$ function definied in Equation (4.8) which takes a $buffer \in Buffers$ and returns 1 when the buffer shall be chosen by the consumer or (otherwise) 0.

$$select_{xchg}(buffer) = \begin{cases} 1 & \text{buffer is explicitely marked as for this consumer} \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

Furthermore it is possible to define a function $progress_{buffer} : Buffer \to Progress$ which returns the progress $p \in Progress$ stored inside a buffer $b \in Buffers$ as $progress_{buffer}(b) = p$.

That makes it possible to retrieve the progresses of an *Xchg* operator by finding the minimum over all in the buffers ($Buffers$) stored progress values. Note that using that minimum as progress might not be always monotonic. Consider the following two cases:

- No progress values are stored in $Buffers$ i.e. no visible (by the *Xchg* operator) progress has been made. Therefore no minimum can be found and the result shall be $nil = [0,0] \in Progress$.
- All producers already completed their processing work and in the current step there are no stored progress values too and the result would be nil too. This would violate the monotonicity property.

As workaround a consumer can store the last progress $progress' \in Progress$ returned and compare the above considered result $current_{progress} \in Progress$ and computes the maximum of both (i.e. $max\{progress', current_{progress}\}$ as defined above). This ensures monotonicity of $progress(op) \in Operator_{buffering}$ and leads to the formulas in Table 4.4.

Table 4.4.: Buffering operators

$$
\begin{aligned}
progress(Reuse_{side=Produce}) &= progress(left)\\
progress(Reuse_{side=Consume}) &= progress(producer(Reuse))\\
progress(XchgUnion) &= max\Big\{progress',\\
&\quad [0, advancement(\min_{b\in Buffers} progress_{buffer}(b))]\Big\}\\
progress(XchgBroadcast) &= max\Big\{progress',\\
&\quad [0, advancement(\min_{b\in Buffers} progress_{buffer}(b))]\Big\}\\
progress(XchgHashSplit) &= max\Big\{progress',\\
&\quad [0, advancement(\\
&\quad \min_{(b\in Buffers)\wedge(select_{xchg}(b)=1)} progress_{buffer}(b))]\Big\}\\
progress(XchgRangeSplit) &= max\Big\{progress',\\
&\quad [0, advancement(\\
&\quad \min_{(b\in Buffers)\wedge(select_{xchg}(b)=1)} progress_{buffer}(b))]\Big\}
\end{aligned}
$$

4.8. Implementation

The progress (value) $p = (stage, advancement) \in Progress$ is implemented as a structure with the size of a native machine word. In that structure *stage* is implemented as an unsigned integer with a length of 32 bit and is therefore limited to $2^{32} - 1$ as a maximal stage number (with 0 being the first stage number). The field *stage* represents $stage(p)$ in the above described model. The progress inside a stage ($advancement(p)$) is implemented as a 32-bit unsigned integer. Its range is limited from 0 to $2^{32} - 2$, because $2^{32} - 1$ is reserved for a purpose of representing invalid progress values, which is the case when - for example - finding the minimal progress over all *Xchg* buffers (of a *Xchg* operator is not possible.

Progress estimation function: The - above described - progress estimation function $progress(op)$ with $op \in Operator$ being the top-most operator of a given QEP (whichs progress shall be estimated). Is implemented as a part of the in, Vectorwise used, operator interface, which can be seen partially in Listing 4.1.

```c
struct Operator {
  /* ... */

  /** fill the vectors with more tuples */
  int (*next)(struct Operator*);

  /** tuple counters */
  int produced, processed;

  /* ... */

  /** Estimates the progress of the operator */
  ProgressEstimate (*get_progress)(struct Operator*);
}
```

Listing 4.1: Implementation of a progress estimation function $progress(op)$ where $op \in Operator$

4.9. Evaluation

The presented and implemented progress estimation will be analyzed in following with the aim to show that it has some of the - in Chapter 4 explained - desirable properties. First the linearity of the progress estimation scheme will be analyzed over the set of TPC-H queries. Afterwards it will be proven that this progress estimation scheme is monotonic.

4.9.1. Linearity

As explained in Chapter 4 linearity of the estimated progress (of a QEP) is a desirable property. In order to analyze the linearity of the progress estimation measurements were done. Each measurement was done with the scheduler code base and the following changes

- Queries were run without parallelism
- The time slice were set to the minimum time slice ($\approx 20ms$)
- Logging progress estimates on yield

That implies that at least after $\approx 20ms$ a progress estimate is logged.

Projected progress: In order to visualize the estimated progress it is helpful to normalize the two-dimensional progress $p \in Progress$ into a one-dimensional progress $n \in \mathbb{Q}$. This has been done by applying function $n = proj(p)$ where

$$proj(p) = 100\% \cdot stage(p) + 100\% \cdot advancement(p)$$

and will be referred as *projected progress*.

Stage change: After changing the stage the progress is calculated using other (compared to the previous meaurement) values. For example in case of an *Aggr* operator the previous stage's progress is calculated using *Aggr*'s child operator. After changing the stage the progress of the *Aggr* operator is calculated based on the number of tuples produced and the number of tuples stored - according to the formula in Table 4.3. This can lead to a changed slope in the projected progress $proj(p)$, caused by the changed *advancement* part of the estimated progress. This is expected non-linear behaviour for this kind of progress estimation and could be damped e.g. by filtering with a low pass.

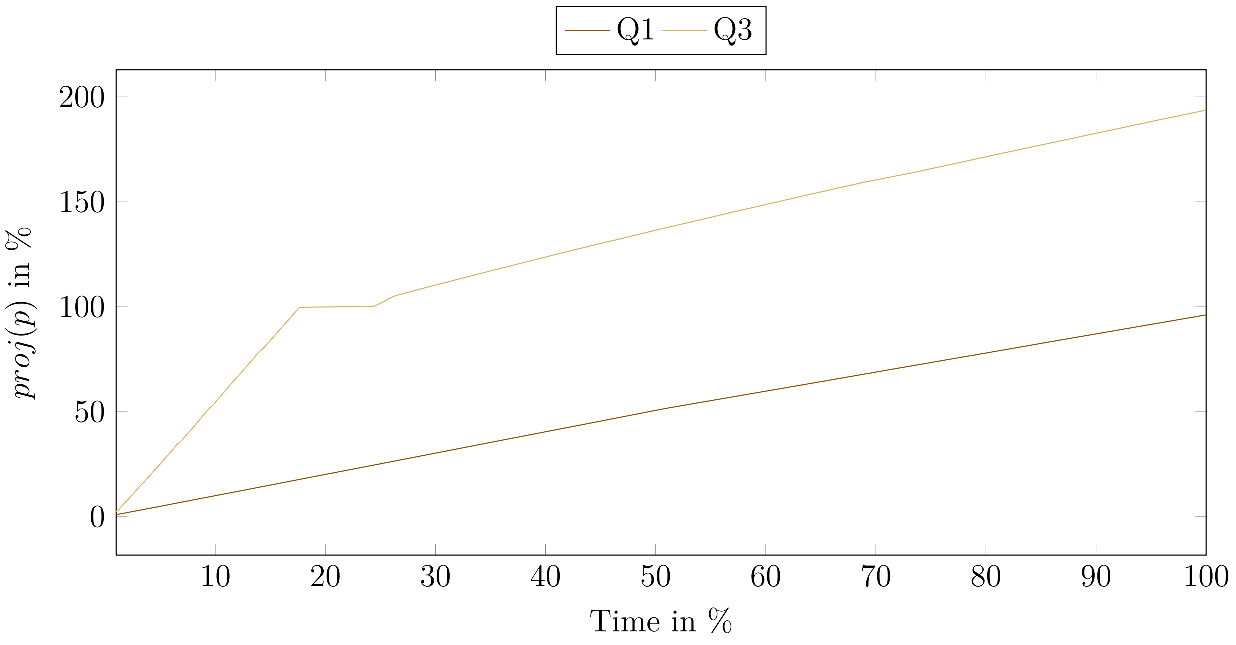
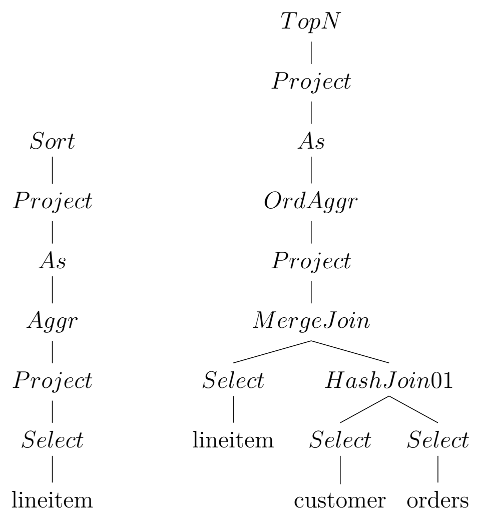


Figure 4.5.: Projected progress of Q1 and Q3

Q1: Expected from the QEP - visualized in Figure 4.6(a) - would be 3 stages (0, 1 and 2), but visualized - in Figure 4.5 - is only one ($0\% \leq proj(p) < 100\%$). That stage must be the first stage (i.e. stage 0) where the *Aggr* operator consums its input tuples. According to the profiling information gathered this - time spent in *Aggr* operator and its subtree in the QEP - takes ≈ 100% of the whole query time. Further this implies that the other stages will take ≈ 0% of the whole query's time and therefore are likely not to be measured and also likely not to be visible.



(a) Q1 (b) Q2

Figure 4.6.: QEPs of Q1 and Q3

Q3: In Figure 4.5 can be seen that in the first $\approx 16\%$ query time the projected progress goes linearly to 100% and stays at 100% for 10 $\approx$ % of query time. According to profiling information gathered $\approx 16\%$ of query time is spent during *TopN consume & HashJoin01 build & Scan lineitem*. The progress is determined by the maximum of the progress of the left and the right child, according to Table 4.2. Both children may run concurrently (from the view of the progress estimation).

- On the right side there is the *HashJoin01* operator building its VHT involves $\approx 16\%$ of the query time.
- Further on the left side scanning (and decompressing) *lineitem* and selecting tuples takes $\approx 27\%$ query time.

Assuming that both sides start at the same time then the right-hand side will show a completed stage ($[stage, 1]$) after $\approx 16\%$ query time and therefore a $proj(p)$ which is a multiple of 100% ($\frac{proj(p)}{1\%}$ mod $100 = 0$). In combination with the estimation formula of the *MergeJoin* operator, the maximum of both progresses (left-hand side and right-hands side progress) is taken as progress of the *MergeJoin*, the *MergeJoin* operator reports a completed stage (here: 100%).

Further the rest of query time ($\approx 84\%$) is spent in the next stage (stage 2) of query execution. According to the QEP in Figure 4.6(b) this can only be one of these stages:

- *TopN consume & HashJoin01 probe* or
- *TopN produce*

According to the profiling information this cannot be the latter one (namely *TopN produce*), because ≈ 0% of query time is spent there. Implying that the time must be spent the stage where *TopN consume & HashJoin01 probe* takes place which therefore reports its progress.

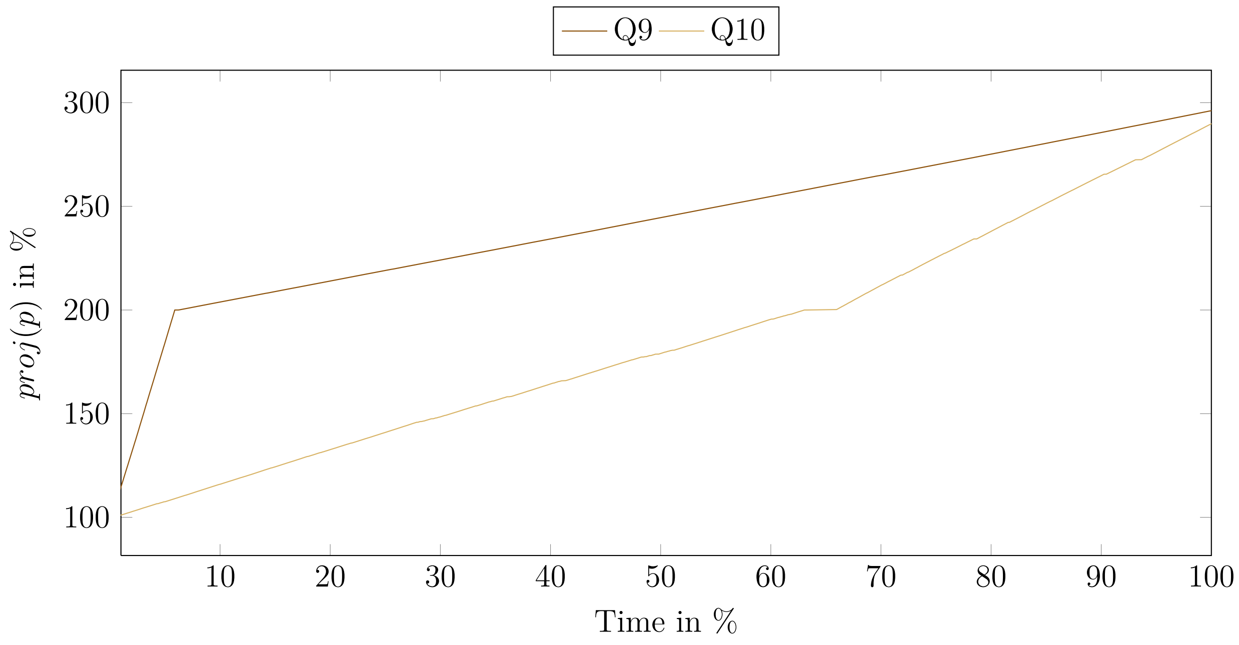
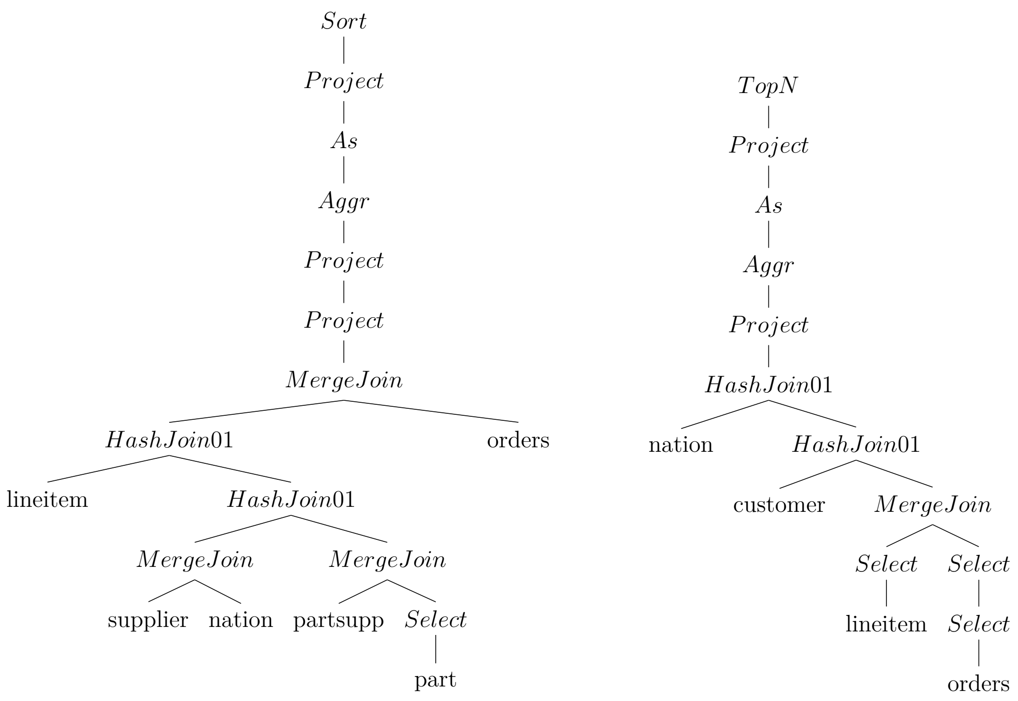


Figure 4.7.: Projected progress of Q9 and Q10

Q9: According to the QEP - visualized in Figure 4.8(a) - Q9 has the following stages:

1. *Sort build & Aggr build & HashJoin build & HashJoin build*,
2. *Sort build & Aggr build & HashJoin build & HashJoin probe*,
3. *Sort build & Aggr build & HashJoin probe*,
4. *Sort build & Aggr produce* and
5. *Sort produce.*

According to the profiling information ≈ 0% query time is spent in the first stage *Sort build & Aggr build & HashJoin build & HashJoin build*, implying that this stage is not visualized in the plot in Figure 4.7. The following stage *Sort build & Aggr build & HashJoin build & HashJoin probe*, consumes ≈ 5% query time where as this is the first stage visible in the plot (Figure 4.7) where $proj(p)$ steadily increases from 100% to 200%. The stage *Sort build & Aggr build & HashJoin probe*, which follows on the previous stage, where ≈ 95% query time is spent and thus visible in the plot where $proj(p)$ increases from 200% to 300%. All other stages do not consume a considerable fraction of query time - according to the profiling information - and therefore are not visible in the plot (Figure 4.7).



(a) Q9 (b) Q10

Figure 4.8.: QEPs of Q9 and Q10

Q10: The projected progress of TPC-H Q10 which uses the QEP shown in Figure 4.8(b), is plotted in Figure 4.7. It can be seen that the projected progress $proj(p)$ increases starting from 100% until 300%. This indicates that the first stage (deepest *HashJoin01* build) is skipped which can be backed by the time spent on that stage which is $\approx 0\%$ of the whole query time - according to the gathered profiling information.

The second stage, where $proj(p)$ increases from 100% up to 200%, is where the probe phase of the previously mentioned *HashJoin01* operator and the build phase of its parent operator takes place. Further also this can be backed by the profiling information, because $\approx 65\%$ of query time is spent there - as also visualized in Figure 4.7 (from 0%) until $\approx 65\%$ of query time.

The third stage, which according to the plotted projected progress, takes $\approx 35\%$ of query time (starting from ≈ 65 query time). Acoording to the profiling information during the build phase of the *Aggr* operator which is a stage at its own.

Further according to the profiling information the other stages (e.g. *TopN* produce, *TopN* build & *Aggr* produce) take no considerable time.

Q17: The QEP of Q17, as visualized in Figure 4.10(a) suggests 5 stages whereas the projected progress - visualized in Figure 4.9 - only shows 2 stages. According to the

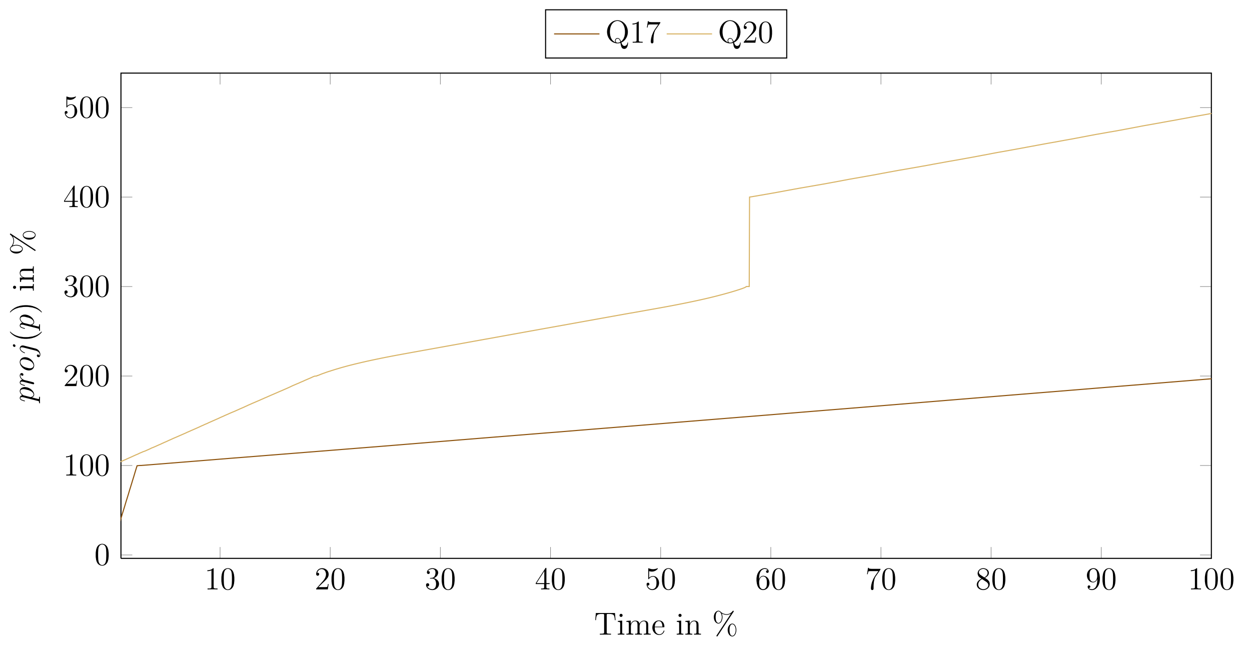


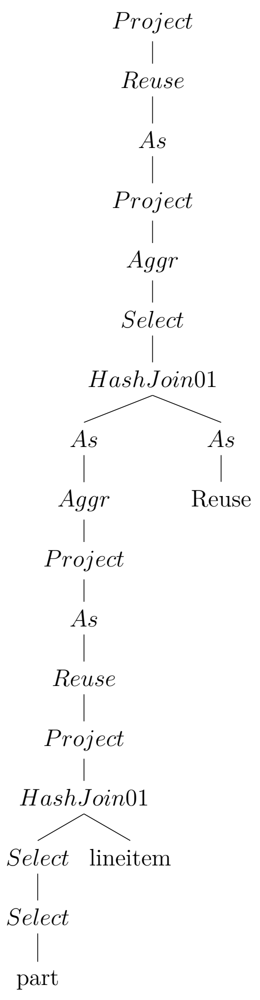
Figure 4.9.: Projected progress of Q17 and Q20

profiling information gathered $\approx$ 100% of the query time are spent in the lower *Aggr* operator and its subtree. Hence stages introduces by the operators above are not visible in the plot. This reduces the number of the potentially visible stages to the following ones:

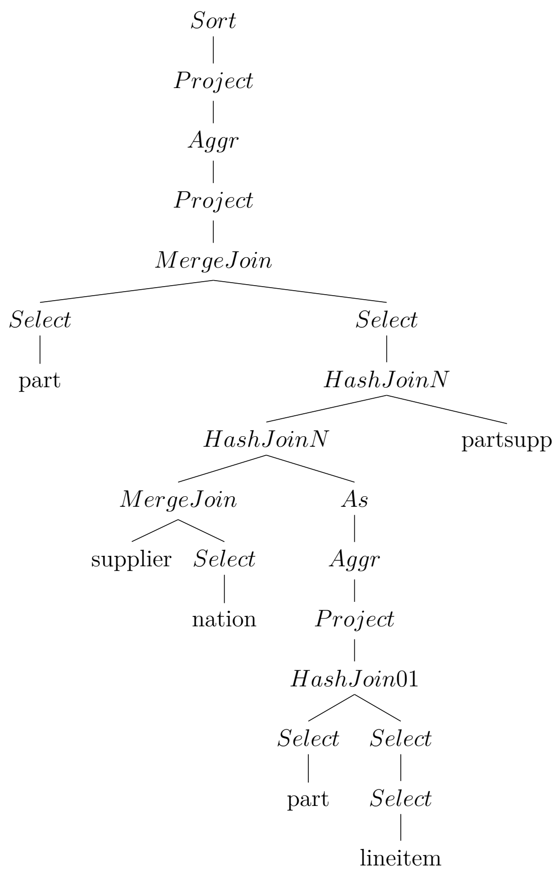
- The first stage of Q17 is the stage where *HashJoin01*'s VHT is built. According to the profiling information gathered this takes $\approx$ 2% of Q17's response time, as it is visible in Figure 4.9.
- In the following stage the tuples are probed through *HashJoin01*'s VHT and are consumed by the *Aggr* operator, which takes $\approx$ 98% of the query response time according to the profiling information.
- The following stage, where *Aggr*'s produce and upper *HashJoin01* builds its VHT, is not visible.

Q20: Another query's projected progress is visualized in Figure 4.9: Q20. The plot suggests that Q20 has the five stages with the first stage being skipped. Further the QEP - visualized in Figure 4.10(b) - suggests 7 stages. The stages except for the QEP below the top-most *Aggr* are listed with their percentage of query time (according to the profiling information gathered), because - according to the profiling information - $<$ 1% query time is spent in the (top-most) *Aggr* and above:

1. The stage where the bottom-most *HashJoinN*'s build phase takes place where $\approx$ 0%



(a) Q17



(b) Q18

Figure 4.10.: QEPs of Q17 and Q20

query time is spent,

2. The stage where the bottom-most *HashJoin01*'s build phase takes place where ≈ 18% query time is spent,
3. In the stage where the bottom-most *HashJoin01*'s probe phase and bottom-most *Aggr*'s build phase takes place, where ≈ 48% query time is spent,
4. The stage where the bottom-most *Aggr*'s produce phase and the bottom-most *HashJoinN* probe-phase takes place, consumes ≈ 0% query time and
5. The stage where the top-most *Aggr* build phase happens, where the rest of the query time is spent.

4.9.2. Monotonicity

As can be seen in Figure 4.5, Figure 4.7 and Figure 4.9 , the projected progress is monotonic increasing. Further it is possible to prove that the progress $p \in Progress$ returned by $p = progress(op)$ for a QEP represented by the top-most operator $op \in Operator$ is monotonic increasing. That means that for a given operator $op \in Operator$ two evaluations of $progress(op)$, with no loss in generality, the first evaluation produces $a \in Progress$ and the second $b \in Progress$, $a \leq b$ will always be a tautology. This means that the progress retrieved by $progress(op)$ cannot ”go backwards”.

Proof. Assuming two progresses p and p' where $p \in Progress$ was measured before $p' \in Progress$. The goal is to show that $p \leq p'$ is a tautology.

This will be proven via induction over height h of the QEP (which forms a tree).

In the case of binary operators it is necessary to define $right, right', left, left' \in Progress$ where:

- $right/right'$ represents the progress of the subtree on the right-hand side in the first/second measurment.
- $left/left'$ are representing the left-hand side subtrees in the first/second measurment.

Transformation: It is necessary to eleminate *Reuse* operators which can be done by replacing the consumer side ($Reuse_{side=Consume}$) by the subtree of the producer side ($Reuse_{side=Produce}$) S. That transformation is valid because it cannot change the monotonicity, because when $progress(S)$ is monotonic, the same has to be valid for a copy of S.

Base case: In the base case ($h = 1$) the QEP can only consist of a *Scan operator*. According to the defined progress the following is valid under the assumption that the totals do not change (i.e. $tuples_{total} = tuples'_{total}$):

$$\begin{aligned}
(p \leq p') &\xLeftrightarrow{\text{apply formula}} \left([0, \frac{tuples_{produced}}{tuples_{total}}] \leq [0, \frac{tuples'_{produced}}{tuples'_{total}}]\right) \\
&\xLeftrightarrow{stage(p)=stage(p')} \left(\frac{tuples_{produced}}{tuples_{total}} \leq \frac{tuples'_{produced}}{tuples'_{total}}\right) \\
&\xLeftrightarrow{tuples_{total}=tuples'_{total}} (tuples_{produced} \leq tuples'_{produced})
\end{aligned}$$

As $tuples_{produced}$ (and $tuples'_{produced}$ too) is a counter in the *Scan operator* which is incremented for each tuple flowed through.

- In case the *Scan operator* has not produced any tuple between the two progress measurements: $tuples_{produced} = tuples'_{produced}$ is valid.
- When the operator has produced tuples $tuples_{produced} < tuples_{produced}$ is valid.

Either one of both cases has to be fullfilled and therefore $tuples_{produced} \leq tuples_{produced}$ is true.

Induction step: Using the base case is is possible to do the induction step (from tree with height $h-1$ to h) in the following way:

By using the operator partitioning - described in Section 4.3 - all operators can be split into four disjoint sets:

- *Scan* operators can only exist on subtrees with height $h = 1$ and are not considered in the induction step.
- *Streaming* operators.
  - The operators *Select*, *Project* and *OrdAgg* are trivial cases.
  - In case of *MergeJoin* monotonicity can be proved like described in the following: Assuming that $left \leq left'$ and $right \leq right'$ holds for a subtree of height $h-1$ (i.e. children of the operator), it is possible to make to following implication:

    $$((left \leq left')) \wedge (right \leq right')) \Rightarrow \Big(max\{right, left\} \leq max\{right', left'\}\Big)$$

    which turns

    $$\Big(max\{right, left\} \leq max\{right', left'\}\Big) \overset{\text{apply formula}}{\Longleftrightarrow} (p \leq p')$$

    into a tautology.
- *Blocking* operators, as described in Section 4.6, can have different phases. These phases get a different stage numbers in the progress. As described in Section 4.2 these phases are also ordered (and get a stage number according to their order). Further a progress with a smaller stage number is always smaller as defined by Equation (4.1).

  That implies that it is only left to prove the monotonicity inside the same phase/stage.

  The operators *Aggr* and *Sort* (the *Sort* operator only has slightly different names for the phases) can be handled in the following way:

  - The phase $G = build$ is analogue to *Select* and is therefore monotonic.
  - The phase $G = produce$ can be handled as described in the following:

$$(p \leq p')$$

$$\overset{\text{apply formula}}{\Longleftrightarrow}$$

$$\Big(\Big[stage(progress(left)) + 1, \frac{tuples_{produced}}{tuples_{in\ hash\ table}}\Big]$$

$$\leq$$

$$\Big[stage(progress(left')) + 1, \frac{tuples'_{produced}}{tuples'_{in\ hash\ table}}\Big]\Big)$$

From the subtree with height $h - 1$ is known that $left \leq left'$.

$$(left \leq left') \Rightarrow \Big(stage(progress(left)) + 1 \leq stage(progress(left')) + 1\Big)$$

With the previous implication it is possible to reduce the formula to the *advancement*:

$$\frac{tuples_{produced}}{tuples_{in\ hash\ table}} \leq \frac{tuples'_{produced}}{tuples'_{in\ hash\ table}}$$

$$\overset{tuples_{in\ hash\ table} = tuples'_{in\ hash\ table}}{\Longleftrightarrow}$$

$$(tuples_{produced} \leq tuples'_{produced})$$

Assuming that the counter $tuples_{produced}$ (and $tuples'_{produced}$) will be only increased, the operators *Aggr* and *Sort* will have monotonic progress estimates.

In case of the *HashJoin* operator which only differs in $G = build$ from *Aggr* (with changing the name of the stage from $G = produce$ to $G = probe$) one can use the same implication as in the previous case and reduce the case to the trivial case (*Select*/*Project* ...).

In the case of the *HashRevJoin* operator it is possibe to prove monotonicity analogue to *HashJoin*, but with a third step $G = produce$ which is analogue to the second step ($G = produce$) of *Aggr*.

- The fourth set of operators are the *Buffering* operators. This set can be reduced into two operators: the *Reuse* operator and the *Xchg* operator.
  - The first one was already eleminated from the QEP by applying a transforma-

tion.

– The latter one abstracts all *Xchg* operators like e.g. *XchgHashSplit* or *XchgUnion*.

  It is possible to abstract the progress of all *Xchg* operators using a buffer selection function $select_{xchg}(b)$ i.e. in case of *XchgUnion* and *XchgBroadcast* one can define: $\forall b \in Buffers : select_{xchg}(b) = 1$.

  By construction the previously returned progress of the *Xchg* operator p is remembered and therefore: $p' = max\Big\{p, [0, a]\Big\}$ with

$$a = advancement(\min_{(b \in Buffers) \wedge (select_{xchg}(b)=1)} progress_{buffer}(b))$$

  This implies $p \leq p'$.

With the previous statements it was proved that the - in this chapter - presented query progress estimation is monotonic.

□

In the following chapter this progress estimation will be used to provide a scheduling strategy that facilitates context specific information (compared to the OS scheduler).

5. User-level scheduler

As mentioned in Section 2.2 Vectorwise uses static parallelism during query execution which leads to a number of problems. In the following it will be assumed that the level of parallelism used is less or equal to the number of cores available. One of issues is the disability to react to dynamic workload changes. Such dynamic workload changes can occur on intra-query level where for example one or more parallel stream(s) suddenly complete their work and therefore a full utilization of all cores cannot be guaranteed. Hence linear scalability cannot be achieved.

Further it will be claimed that static Volcano-model parallelism can be made dynamic using virtually more threads than cores and carefully balancing these threads across the cores.

Overallocation: This problem is assumed to be solvable efficiently through overallocation i.e. using more threads and using those extra threads to balance out the different processing speeds of each thread. In the best case all threads would finish their work at the same time and there will be no skew (except for the serial main (kernel) thread, which collects the data from the parallel (kernel/user-level) threads).

5.1. Approach

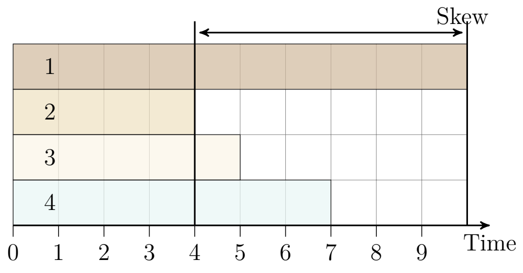
Overallocation in this context means that the system uses more threads than the hardware can process in parallel i.e. on the 64 core system e.g. 80 threads. This involves distributing the data over virtually more threads which decreases the amount of data to be processed per ULT[1] and therefore the time needed, until a bound.

These extra threads can be used to close the gaps. These gaps are created by threads which are completing earlier and would therefore produce *skew*. Consider the following example.

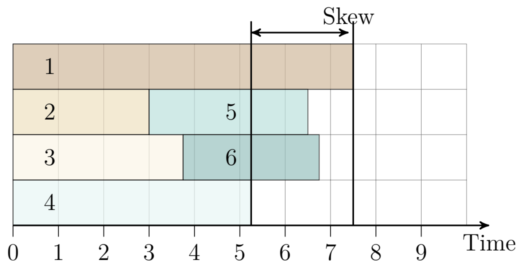
[1]User level thread

Table 5.1.: Example workload without overallocation and with 50 % overallocation

Thread	Time without overallocation	Time with 50 % overallocation
1	10	7.5
2	4	3
3	5	3.75
4	7	5.25
5	-	3.5
6	-	3



(a) no overallocation



(b) 50 % overallocation i.e. two additional threads

Figure 5.1.: Example with 4 threads and 6 threads (50% overallocation)

Example: Imagine a workload as given by Table 5.1 in which 4 threads are running on a theoretical system with 4 processors. Further it can be seen that there is execution skew - as visualized in Figure 5.1(a). Further given that the example workload is parallelizable using 6 threads on the same (4 processor) system. Then it is possible - as visualized in Figure 5.1(b) - to balance these across the processors in order to keep the processors utilized. Further Figure 5.1(b) shows that this combination of scheduling and overallocation reduced (execution) skew.

5.1.1. Time to completion

Based on the progress of a QEP, it is possible to estimate the time to completion (of a stage).

It was assumed that the speed of an operator (and its subtree) will change temporarily which might happen when a QEP runs on a different processor and might be caused by serveral reasons cold caches, NUMA effects i.e. non-local memory access, ...

Windowing: Therefore 2 windows consisting of the last $n = 4$ samples were used, one for recording the progresses and one for the time needed to reach each progress for a QEP

q. It is possible to define a window P_0, P_1, ..., P_{n-1} consisting of the last n progresses made in a time slice and a window T_0, T_1, ..., T_{n-1} consisting of the last n time units the task has run inside its time slice. So that the speed $v(q)$ for a QEP q can be calculated over these windows as in Equation (5.1).

$$v(q) = \frac{\sum_{i=0}^{n-1} P_i}{\sum_{i=0}^{n-1} T_i} \tag{5.1}$$

Time to completion: With the help of the speed $v(q)$ and an estimated progress $progress(q)$ it is possible to define the time to completion (of a stage) ttc with Equation (5.2).

$$ttc(q) = \frac{1 - advancement(progress(q))}{v(q)} \tag{5.2}$$

Note that $\forall p' \in Progress$: $advancement(p')$ has an upper bound which is, by definition, 1 (see Section 4.2) and that p' is monotonic increasing. This implies that constant 1 denotes the highest value $advancement(p)$ can reach. Implying that $ttc \geq 0$ is a tautology.

5.1.2. Workload metric

Based on the estimated time to completion in Section 5.1.1, it is possible to construct a workload metric, which represents an estimation of the work left in a stage: For a given set of QEPs to run Q it is possible to build such a metric W by summing the estimated time to completions for all QEPs, which results in Equation (5.3).

$$W(Q) = \sum_{q \in Q} ttc(q) \tag{5.3}$$

With the help of this metric W it is possible to detect, for two sets of QEPs A and B, underload and overload situations. A is underloaded compared to B when A has less work left than B or using Equation (5.3) when

$$\sum_{q \in A} ttc(q) < \sum_{q \in B} ttc(q).$$

Futher A is overloaded compared to B when

$$\sum_{q \in A} ttc(q) > \sum_{q \in B} ttc(q).$$

This metric will be used in the following in order to decide whether a scheduler accepts to run a QEP which is located on a remote NUMA node or not.

5.1.3. QEP decomposition

In Vectorwise the *Xchg* operators provide a natural splitting point for seperating producer and consumer sides where - from the view of the *Xchg* operator the producer threads are running the QEP below the *Xchg* operator and the consumer threads the QEP above.

In order to keep producer and consumer sides active, independent of the per-thread scheduling strategy, it was necessary to split the QEP at these points i.e. at each *Xchg* operator.

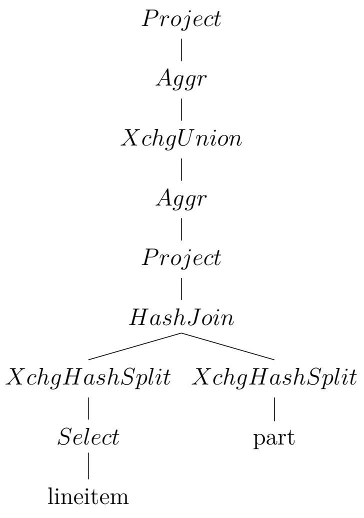


Figure 5.2.: Example QEP

As visualized in Figure 5.2, a parallel QEP can be decomposed into parallel running query pipelines (also QEPs) by splitting at these *Xchg* operators (here these are *XchgUnion* and *XchgHashSplit*). This leads to QEPs as visualized in Figure 5.3.

Each of those decomposed QEPs are part of a system-wide QEP-pool and will be scheduled by the alogrithm described in Section 5.2.5.

These QEPs can be run by multiple threads (the top-most QEP cannot be run in parallel). which is usually a higher number than the number of kernel threads in the worker (kernel) thread pool, except the rewriter decides only to introduce a smaller degree of parallelism.

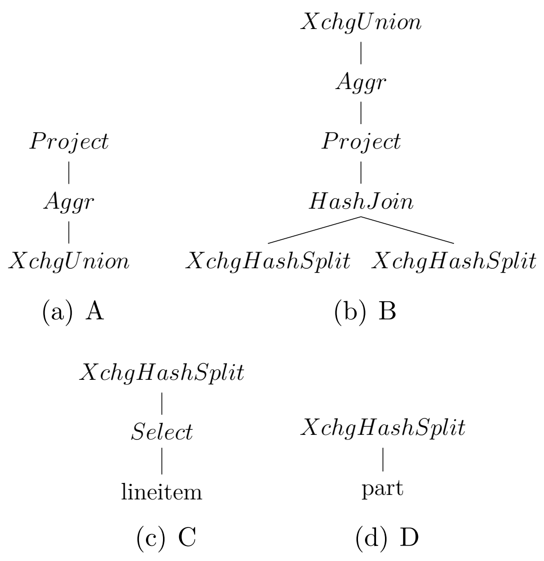


Figure 5.3.: Example QEP

5.1.4. Scheduling

In case that the currently running thread may cooperatively yield the processor or are waiting on a synchronization primitive (i.e. mutex or condition variable), it is necessary to make a decision which thread to run next or whether the processor should idle instead.

Stage-based scheduling: As explained in Section 4.2, the evaluation of a query/QEP can be partitioned into stages which are active sequentially. Further skew is possible at the end of each stage. This implies that it is necessary to achieve balance at the end of each stage.

Greedy scheduling: As described in Chapter 4, it is possible to estimate the progress of a QEP and to, based on that, calculate an estimated time to completion (see Section 5.1.1). With the help of the time to completion it is possible to determine which QEPs are slower or lagging behind, because these have a higher time to completion.

In order to balance all threads equally it is necessary to accelerate these (slower or lagging behind) query processing threads, represented by their QEPs, by scheduling them more often. This leads the a greedy scheduling strategy where the next QEP, to be processed, is the one with the maximal time to completion.

Adaptivity: Further a concept of time slices forces an update of the schedule and the gathered estimated progress. The size of these time slices decreases with the time to completion. This leads towards more fine-grained balancing (i.e. small time slices) at the end of stage while having coarse-grained balancing (i.e. large time slices) in the "middle" of a stage. This helps to tackle inaccurate time to completion estimates and therefore sub-optimal scheduling decisions. Further this helps to reduce the inefficiency introduced by very small time slices caused by the number of scheduling decisions, cache misses, context switches etc.

5.2. Implementation

This section presents the implementation of the user-level scheduler starting with the implementation of overallocation in Vectorwise.

5.2.1. Overallocation

As explained in Section 2.2, Vectorwise DBMS uses an cost-based optimizer inside the rewriter which is used to introduce parallelism into the QEP by adding *Xchg* operators (in combination a level of parallelism specified) and specialized rules for each operator.

This concept can also be used to implement the overallocation by factor o by the following changes:

- Increase the number of threads checked by factor o.
- Adapt the cost model to make higher degrees of parallelism cheaper.

5.2.2. Thread API emulation

Vectorwise has an API that abstracts

- threads,
- mutexes,
- shared-read-exclusive-write locks and
- condition variables.

In combination with user-level scheduling this API has to provide an equivalent implementation which is described in this section.

Mutex

Mutex (mutual exclusion) protects a critical section against concurrent accesses and provides two function *mutex_lock* and *mutex_unlock*. The function *mutex_lock* starts a critical section protected against concurrent access and *mutex_unlock* ends it.

Implementation: The implementation was inspired by [Inc14]. In order to improve the performance in the uncontented/less contented case the implementated mutex waits a short period of time actively which avoids scheduling and context switching overhead in the mentioned case when there is less contention.

After waiting that short period and not being in the critical section, the ULT will be put to sleep until it will be woken up in *mutex_unlock*.

Condition variables

Condition variables allow to put a thread to sleep while waiting for a variable to change or a resource to become available. It provides the following two basic functions:

- *cond_wait*, which *might* wait until the thread will be woken up by one of the following functions:
- *cond_broadcast*, which notifies all waiting threads and
- *cond_signal*, which notifies one waiting thread.

cond_wait: A user-level thread calling *wait* causes a *context switch* (see Section 5.2.4) after that it is moved into the condition variable's queue.

cond_broadcast: When *cond_broadcast* is called *all* threads that are in the queue of the condition variable are made runnable again i.e. they were enqueued into the appropriate queue of the scheduler. Note that a condition variable is always used in combination with a mutex. This implies that potentially many threads are made runnable again which all try to lock the mutex (*mutex_lock*) where only one thread can succeed, the others are actively waiting for a short period and then put to sleep to be woken up by *mutex_unlock*. This is known as the *thundering herd problem*. In case of *XchgHashSplit* with many threads on both (producer and consumer side), this was found out to be a performance problem.

This makes it possible to implement the following optimization that works around the explained thundering herd problem in a way like the Linux kernel & pthreads solved it: In case of a contented mutex, it is possible to move all threads from the condition variable's queue into the queue of the mutex, because a thread leaving the critical section, calls *mutex_unlock* which makes one thread from the mutex' queue runnable again. In case of an uncontented mutex, only one thread has to be made runnable again which makes the mutex contented and the optimization's contented case can be used again.

cond_signal: A condition variable also makes it possible to only wake one thread waiting on it. This is handled by *cond_signal*. Further *cond_signal* is handled like *cond_broadcast*.

5.2.3. Fibers

"A *fiber* are [light-weight] unit of execution that must be manually scheduled by the application" [Mic14]. This concept was used as an abstraction of low-level context switching in order to hide the real implementation of context switching behind an interface.

Creating fibers: In general it is possible to create a fiber in two ways:

- It is possible to convert parts of the current thread of execution into a fiber or
- to create a fiber from functions.

Switching: It is possible to switch from one fiber (*fiber A*) to another fiber (called *fiber B*), which involves saving *fiber A*'s registers, loading *fiber B*'s registers and jumping to *fiber B*'s instructions.

Implementations

Through the abstraction provided by fibers it is possible to implement context switching between threads of execution on different ways:

- POSIX[2] *swapcontext*
- *setjmp/longjmp*
- using multiple kernel threads and control the ability to run over *condition variables*
- ...

[2]Portable operating system interface

In the frame of this thesis the implementation over the POSIX *swapcontext* and *setjmp/longjmp* was done.

POSIX createcontext/swapcontext allows the create execution contexts (*createcontext*) and to switch between them (*swapcontext*). The implementation using POSIX *swapcontext* had the disadvantage that the overhead of each switch between fibers costs at least two *system call*s. Further it was noticed that it produced contention in the kernel itself when a high number of currently running kernel thread called *swapcontext*.

setjmp/longjmp: Because of these disadvantages were undesirable for a solution which might switch many times between fibers and possibly many occur concurrently switches (between fibers), an alternative method has to be found. The C standard library provides two functions that allow to change the execution flow in a way that makes them usable for fibers: *setjmp* and *longjmp*. The function *setjmp* stores the current execution context into a buffer. The function *longjmp* allows to restore an execution context from a buffer created by *setjmp*. The functions *setjmp* and *longjmp* have the disadvantage that they do not save and restore the stack pointer. [Vyu14] provides a faster solution by mixing *setjmp/longjmp* and *swapcontext*. *Swapcontext* and its utility functions can be used to create an execution context (using *createcontext*), switch to it (using *swapcontext*) and switch back using *setjmp/longjmp*. This lead to an environment in which it is possible to use *setjmp/longjmp* to switch between fibers without losing or using the wrong stack pointer while keeping the property of cheaper switching between the fibers (not involing at least two system calls).

Microbenchmark: In order to compare the performance of the variants (*POSIX createcontext/swapcontext* and *setjmp/longjmp*) a microbenchmark was created. This microbenchmark consists of three fibers:

- The *main* fiber which firstly switches once to
- The fiber *ping* switches in a loop to
- Fiber *pong* which switches in a loop back to fiber *ping*.

The partial source code can be found in Listing 5.1 (the full source code is available in Appendix A). Note that this piece of source code misses the actual implementation of fibers. Further note that the function *ticks()* returns the current cycle counter on the CPU (i.e. *rdtsc*). Two versions of the microbenchmark program were created, one for each implementation. Both were compiled with GCC in Version 4.4.7 using optimization level 3.

```
#define N (1024*1024)
fiber_t ping_fiber, pong_fiber, main_fiber;
uint64_t ping2pong = 0, pong2ping = 0, count = N;
uint64_t diff[N];

void ping() {
  while (count > 0) {
    ping2pong = ticks();
    switch_to_fiber(&pong_fiber, &ping_fiber);
    diff[count] = ticks()   pong2ping;
    count = count   1;
  }
  switch_to_fiber(&main_fiber, &ping_fiber);
}

void pong() {
  while (count > 0) {
    pong2ping = ticks();
    switch_to_fiber(&ping_fiber, &pong_fiber);
    diff[count] = ticks()   ping2pong;
    count = count   1;
  }
  switch_to_fiber(&main_fiber, &pong_fiber);
}

int main() {
  memset(&main_fiber, 0, sizeof(fiber_t));
  create_fiber(&ping_fiber, &ping);
  create_fiber(&pong_fiber, &pong);

  switch_to_fiber(&ping_fiber, &main_fiber);

  uint64_t sum = 0, max = 0, min = 42424242;
  for(i=1024; i<N; i++) {
    sum += diff[i];
    count++;
    max = diff[i] > max ? diff[i] : max;
    min = diff[i] < min ? diff[i] : min;
  }

  printf("Mean: %llu\nMinimum: %llu\nMaximum: %llu\n",sum/count,min,max);
  return 0;
}
```

Listing 5.1: Ping-pong microbenchmark

Table 5.2.: Results of the ping-pong microbenchmark in cycles

Variant	Minimum	Mean	Maximum
POSIX createcontext/swapcontext	1267	1279	81722
setjmp/longjmp	111	112	43168

From the results of the microbenchmark in Table 5.2 can be seen that *setjmp/longjmp* allows faster context switching in the average case. Note that the minimal context switching time is close to the average switching time. The extremely high maximal context switching time could be explained by the fact that during the microbenchmark's runtime the Linux kernel preempted the process, because each run took ≈ 2s. Further assuming that system calls, as they are done by *swapcontext*, provide points where the Linux kernel preferably preempts processes and the Linux kernel preempted the process two times inside *swapcontext*, it sounds reasonable that the maximal context switching time of *POSIX createcontext/swapcontext* is twice as high.

5.2.4. Context switching

A context switch is the transition from a user-level thread (or the idle state) to another user-level thread (or to the idle state).

The context switch will be triggered by a user-level thread explicitly yielding the processor which happens through:

- waiting on a condition variable,
- noticing that the user-level thread has exceeded its time quantum,
- calling the *yield* function which directly yields the processor,
- or by a worker (kernel-)thread which is in the idle state to check whether it can run a user-level thread, if that is possible it switches directly to the chosen user-level thread.

Algorithm: This triggers the algorithm visualized in Figure 5.4.

At first the currently running user-level thread - without loss of generality called *thread A* - saves its metrics which is storing its current information (e.g. time *thread A* was actively running, time it spent in a critical section, time it spent waiting on a mutex, ...).

After that the statistics of *thread A* will be updated which means calculating the estimated time to completion based on the estimated progress and the needed time for reaching the current progress.

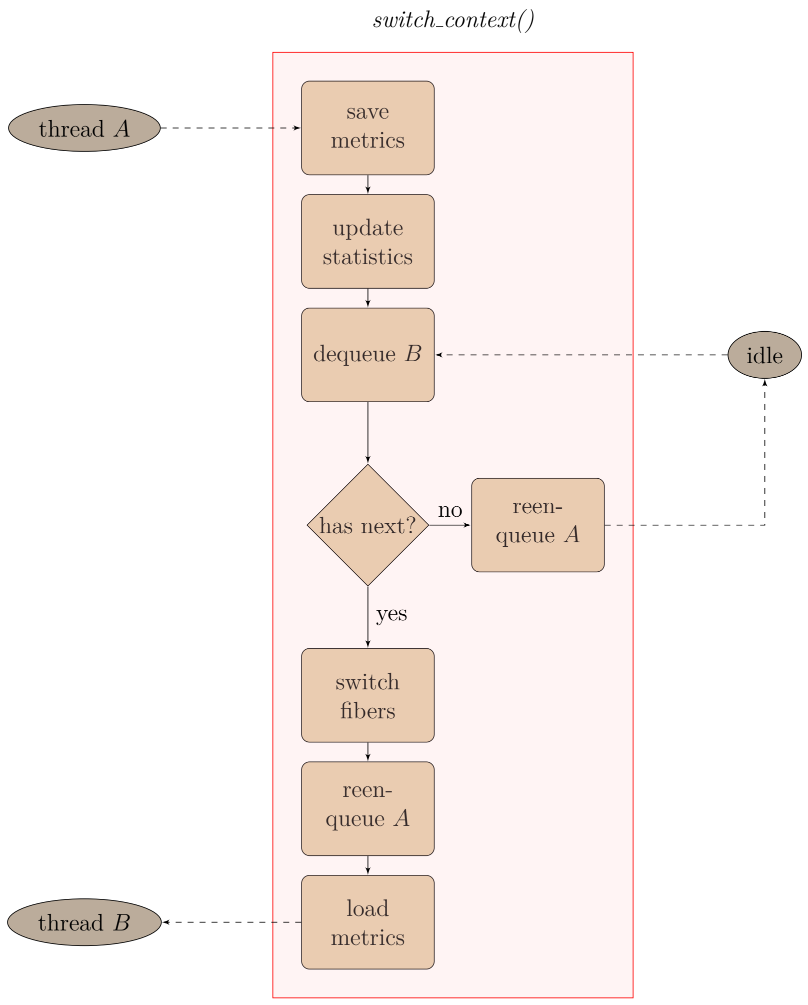


Figure 5.4.: Context switching

Then *thread A* will be enqueued into one of scheduler's queues and the scheduling algorithm will decide which user-level thread to run next.

If the scheduling algorithm was not able to find a user-level thread that is runnable, it will change its state to *idle*, which means sleeping a few milliseconds and then check again.

If a runnable user-level thread without loss of generality called *thread B* - could be chosen, then it will initialize *thread B*'s fiber if necessary (only once at the first start of *thread B*) and switch from the *thread A*'s fiber to *thread B*'s fiber (for switching between fibers, see Section 5.2.3).

After that load the previously stored metrics for *thread B* and *thread B* continues its work.

5.2.5. Scheduling

Basically the scheduler has to decide which user-level thread to run on the kernel thread which is asking for work to do. That problem has a few requirements:

- It should fill the gaps that would cause skew.
- It has to take memory locality in account. That means it should prefer user-level threads for which the memory access would be (mostly) local. That is needed to avoid excessive transferring of remote memory over the shared bus, because the latencies for remote memory access are higher it will slow down a user-level thread and in the worst case, the bandwith limit of the shared bus will be reached and remote memory access latencies will increase additionally.
- Ability to control the granularity of work each user-level thread executes. This allows - on one hand-side - a more efficient execution (using a coarse granularity, involving less context switching and scheduling overhead) while the user-level thread is expected not to directly complete of that chosen granularity. On the other hand-side this allows more fine-grained balancing when user-level threads are close to completion (chosing a low granularity of work).

Algorithm: The scheduler employs a pool of worker (kernel) threads which repeatedly ask for work (when there is none, they sleep for a given period).

When a worker (kernel) thread requests work, it first selects one of the many possible QEPs, after that it selects the (user-level) thread which has the most work left until completion. Note that multiple worker (kernel) threads might be processing different QEPs.

- Split the QEP into sequential parts i.e. without Xchg operators, called *PP*[3] as defined in Section 5.1.3
- such a pipeline can be executed by multiple threads (not the same physical operator pipeline)
- each user-level thread of such a part is spawned inside such a part and are enqueued on the queue of their preferred NUMA node
- each queue is ordered (descending) by the estimated time to completion i.e. it will return the user-level thread with the highest time to completion
- run that thread for an appropriate time slice.

[3]Parallel pipeline

PP data structure: The data structure of a PP can be seen in Listing 5.2. Each PP holds a definite number of priority queues in which references to the runnable threads are stored - one per NUMA node as *queue*. These priority queues are ordered descendingly by the time of completion of the thread i.e. the thread with the highest time to completion will be dequeued first. Such an priority queue exists for each NUMA node (in the PP data structure) where the ULTs are enqueued on the NUMA node on which they are assumed to have local memory access. Note that each priority queue is implemented using a linked-list, because the code of the linked-list allows it to be reused for FIFO[4] queues and a assumed maximal length of the queue of 10 (for 80 ULTs on 8 NUMA nodes).

```
struct ThreadQueue {
  Thread* head;
  Thread* tail;
}

struct ParallelPipeline {
  /* Work left metric for each NUMA node */
  long* work_left;

  /* User level thread queue for each NUMA node */
  struct ThreadQueue* queue;

  /* Pointer to the next PP in the PP list */
  struct ParallelPipeline* next;

  /* Pointer to the previous PP in the PP list */
  struct ParallelPipeline* prev;

  /* ... */
}
```

Listing 5.2: PP data structure

Furthermore the PP data structure maintains statistics about the amount of work to be done on the NUMA-node (*work_left*) which uses the metric from Section 5.1.2 to represent to workload on a NUMA node.

To be able to handle multiple queries or queries with more than one *Xchg* operator (using the decomposition from Section 5.1.3), multiple PPs form a doubly linked-list. Hence pointers to the next (*next*) and previous PP (*prev*) are needed.

Consider the example given in Figure 5.5 which visualizes two queries (α and β) running

[4]First in, first out

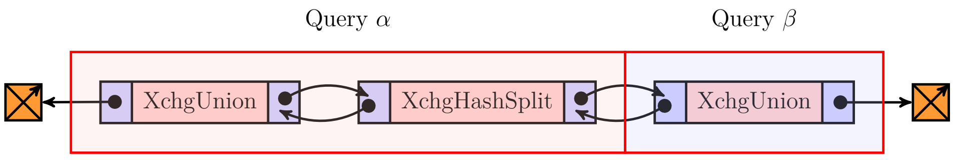


Figure 5.5.: Linked PPs for two queries

in parallel. Query α consists of a QEP that involves two *Xchg* operators in sum. Using the QEP decomposition, described in Section 5.1.3, these two *Xchg* operators can be linked together. In Figure 5.5 these are *XchgUnion* and *XchgHashSplit*. Then the *Xchg* operators of query β are linked in the same way. In the given example there is only a *XchgUnion* operator. Further the linked-lists of query α and β are combined forming the list as visualized in Figure 5.5.

Note that through this the hierarchy formed by queries consisting of *Xchg* operators is flattened into one linked-list.

PP decision algorithm: The first decision the scheduler has to make is which part of a query (PP) it shall run. These PPs contain different sides (producer or consumer) of each *Xchg* operator where all of these sides shall kept active to avoid unneccesary context switching and scheduling.

Therefore the round-robin algorithm is used for deciding which QEP to run next. It is possible that a PP has no runnable threads in that case it is skipped. Note that this is based on the assumption that every *Xchg* operator of each query shall be kept alive in a best-effort fashion by scheduling the PP using round-robin. Further the round-robin algorithm ensures a fair distribution between all PPs.

Thread decision algorithm: The thread decision algorithm - visualized in Algorithm 1 - decides which ULT to run next. For this decision the following sub-decisions have to be made:

a) Shall that ULT be one from the local NUMA node or shall it steal work from a remote (NUMA) node?

b) Which ULT of the chosen NUMA node shall be executed?

In the following these sub-decisions will be answered using the scheduling algorithm (Algorithm 1) where the used functions defined as the following ones:

- The function *FindBestThread* takes a PP p and a NUMA node $node \in Nodes$ and returns the ULT with the highest estimated time to completion as defined in Section 5.1.1. Note the function *FindBestThread* answers sub-decision b). The

```
input  : PP to run ($p$), Set of all NUMA nodes $Nodes$, Current/local NUMA
         node $local \in Nodes$
output: User-level thread to run
1  $best_{local} \leftarrow$ FindBestThread($p$,$local$);
2  $best \leftarrow best_{local}$;
3  foreach $node \in Nodes \setminus \{local\}$ do
4      $curr \leftarrow$ FindBestThread($p$,$node$);
5      Nodes which are stages behind need to be boosted;
6      $a \leftarrow$ (GetStage($curr$) < GetStage($best_{local}$));
7      Use theshold for nodes inside a stage, to avoid needless remote work through
       load balancing;
8      $b \leftarrow$ (WorkLeft($p$,$node$) > $threshold$ +WorkLeft($p$,$local$) ) $\wedge$ (
       GetStage($curr$) = GetStage($best_{local}$) );
9      Find the best of both;
10     $c \leftarrow$ OverwriteBest($curr$,$best$);
11     if $((a) \vee (b)) \wedge (c)$ then
12         $best \leftarrow curr$;
13 end
14 Remove($best$);
15 return $best$;
```

Algorithm 1: Task scheduling decision

implementation of $FindBestThread$ searches linearly in one run for the minimal stage and the maximal estimated time to completion where is the estimated time to completion is only considered when the stages are equal. Note that this implementation is not optimal for large amounts of ULTs per NUMA node, in the latter case a priority queue could be used in order to find such a ULT in time complexity of $O(log(n))$. The result of defining $FindBestThread$ that way is that:

- ULTs that are lagging one or more stage behind are preferred and that
- Inside a stage ULTs with a higher time to completion are preferred.

This method consequently prefers ULTs that are lagging behind according to the estimated progress and the time each ULT took to reach this progress.

- Further $GetStage$ is a function that returns the stage from the last measured progress from the given ULT as defined in Section 4.2 (as *stage*).
- The function $WorkLeft$ returns the workload metric, that is defined in Section 5.1.2, for a given PP p and a NUMA node $n \in Nodes$. $WorkLeft(p,n) = W\Big(\{q \in GetQEPs(p) \mid GetNode(q) = n\}\Big)$ where
  - $GetQEPs$ returns the set of all QEPs in the given PP p and
  - $GetNode$ returns the NUMA node $a \in Nodes$ on which q was running first.

- *OverwriteBest* takes two ULTs (a and b) and decides whether a would be a better choice than b. It is defined in the following way:

$$OverwriteBest(a,b) := \begin{cases} true & \text{if } GetStage(a) < GetStage(b) \\ true & \text{if } (GetStage(a) = GetStage(b)) \wedge \\ & \quad (GetTTC(a) > GetTTC(b)) \\ false & \text{otherwise} \end{cases}$$

  where $GetTTC$ returns the estimated time to completion of a given thread as defined in Section 5.1.1.

- *Remove* removes the given ULT from the queue where it is currently enqueued.

Algorithm 1 works in the way that it first tries to find a ULT located on local NUMA node using *FindBestThread*. Afterwards Algorithm 1 is trying to find a better ULT via the *OverwriteBest* function, where only in following cases ULTs from remote NUMA nodes are considered:

- When it detects that a NUMA node is overloaded. In that case the workload metric of the remote NUMA node is higher than the workload metric of the local NUMA node by at least *threshold*.
- A ULT is lagging one or more stages behind.

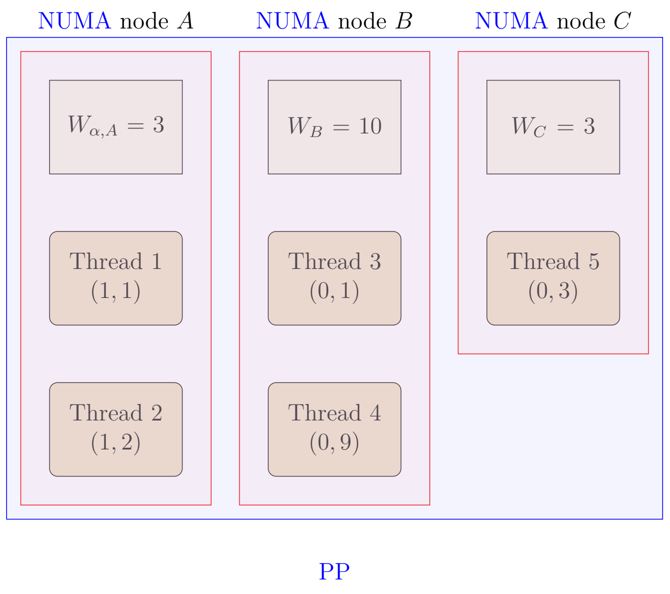


Figure 5.6.: Scheduling example

Scheduling example: Consider the state given in Figure 5.6. Further the visualized PP has 5 threads (*Thread 1*, *Thread 2*, *Thread 3*, *Thread 4* and *Thread 5*) where all threads are distributed over 3 NUMA nodes (A, B and C). Each ULT has a tuple $(stage, ttc)$ assigned where $stage$ defines the last stage measured via the progress estimation and ttc the time to completion calculated.

Now a scheduling decision has to be made for a processor on NUMA node A with $threshold = 5$:

The best choice for running a ULT from the local NUMA node (A) is initialized using the ULT with the minimal stage and maximal time to completion. In this example it is *Thread 2* with a stage 1 and a time to completion of 2.

Afterwards the currently best choice $best$ is initialized to the best choice of the local NUMA node $best_{local}$ which is *Thread 2*.

Now remote NUMA nodes are considered in order to obtain a better scheduling choice than the local choice.

1. In order to achieve this the best choice on the NUMA node B has to be found (via $FindBestThread$) which is in example returns *Thread 4* with $stage = 0$ and $ttc = 9$. Now the two choices - *Thread 2* and *Thread 4* have to be compared: First it needs to be checked whether a ULT from a remote NUMA node has to be preferred because it is lagging one or more stages behind the ULT on the local NUMA node. In this example *Thread 4* is lagging 1 stage behind *Thread 2* because *Thread 2* has $stage = 1$ and *Thread 4* has $stage = 0$ where stages can only increase. Further b would be $false$ because the stages are not equal. Afterwards it need to be decided whether the current best choice shall be overwritten, because the current choice $curr$ (*Thread 4*) is better. Given the fact that *Thread 4* is lagging behind (compared to $best = best_{local} =$ *Thread 2*) it will overwrite the current best choice $best$.

2. Afterwards NUMA node C best choice (*Thread 5*) is compared with the choice from the local NUMA node $best_{local} =$ *Thread 2*. With $stage = 0$ and $ttc = 3$ *Thread 5*, regarding stages, is not lagging behind the local choice. Further the workload of NUMA node A plus $threshold$, $W_A + threshold = 3 + 5 = 8$, is greater than workload of NUMA node C, $W_C = 3$, implying that from the view of NUMA node A NUMA node C appears not overloaded, so that NUMA node A will no consider executing work from NUMA node C. Leading that none of the prequisites for a new best choice is not given. Thus *Thread 5* is ignored in this example.

As with *Thread 5* being the best choice found so far $best$ and all NUMA nodes being considered, the scheduling algorithm shown in Algorithm 1 will run *Thread 5* as next thread. The chosen thread runs until it is getting preempted or blocked, then a new scheduling decision has to be made.

5.2.6. Cooperative multitasking

In order to refresh the schedule and provide fairness between the ULTs it was necessary to implement and assert time slices. These time slices determine an approximate maximal runtime of a ULT until the next scheduling where another ULT is getting a chance to run, which is determined by the scheduling algorithm explain in Section 5.2.5. These time slices are checked in a cooperative way via calling the *cooperate()* function in the QEP.

Time slices: The idea is to make the time slices dependent on the estimated work left, in a way that these are getting smaller when the ULT - for which the time slice is calculate - is getting closer to completion. Due to wrongly estimated time to completion (during first time periods of a ULT) it was also necessary so increase the time slices in a more pessimistic way, so that until a specified time - which the ULT - ran (t_{min}) only the smallest time slices are given.

$$t_{slice} = max\Big\{min\{\frac{work}{l \cdot n}, max_{time\ slice}\} \cdot t_{select}, min_{time\ slice}\Big\} \tag{5.4}$$

Equation (5.4) denotes the used formula to calculate the time slice, where

- $work$ is the amount of work left on the NUMA node,
- n the number of active (user-level) threads,
- l a constant depending on the NUMA locality further defined as

$$l = \begin{cases} 2 & \text{if local} \\ 31 & \text{if remote} \end{cases}$$

- t_{active} the time the (user-level) thread ran,
- $max_{time\ slice}$ the highest possible time slice,
- $min_{time\ slice}$ the lowest possible one and
- t_{select} is defined as

$$t_{select} = \begin{cases} 0 & \text{if } t_{active} \leq t_{min} \\ 1 & \text{otherwise} \end{cases}$$

Cooperation: In order to assert the given time slices, a mechanism of cooperation is needed. This mechanism is realized through a function *cooperate()*. This function checks whether the currently running (user-level) thread is still inside its time slice, otherwise that function will yield back to the scheduler which decides which (user-level) thread to run next. The function *cooperate* is called in every operator producing tuples (e.g. *MScan*, *Xchg*, *Aggr*, ...) and the parallelization operator (i.e. *Xchg*) while consuming tuples. This guarantees that *cooperate()* is called at least once when the results travel up the pipeline, while trying to keep the overhead minimal by calling *cooperate()* only per stride (set of vectors) and only at certain points of query execution.

5.3. Evaluation

This section evaluates the performance of the implemented user-level scheduler, starting with an analysis of the balance that could be achieved in two short-running queries. The second part will evaluate how the user-level scheduler behaves when the parallel-running threads show imbalance. This will be analyzed over a microbench that provides different degrees of skew. Last but not least it will be analyized whether the query response times could be improved over the industry-leading TPC-H benchmark.

5.3.1. Balancing behaviour

In order to analyze the balance that is achieved 2 rather short-running TPC-H queries were taken. These are Q2 and Q12.

Configuration: These queries were run on Vectorwise with implemented user-level scheduling. Further it was configured in a way that 2 NUMA nodes with 4 processors each are used. Each query was run twice:

- One time using $n = 8$ ULTs. In this case the implemented scheduling algorithm provides no advantage, because it has not enough threads available to choose the best one i.e. it can only choose between idle and one ULT.
- Another time using $n = 12$ ULTs, where the user-level scheduler has the possibility to choose between different ULTs.

Note that n represents the level of concurrency.

Q2: Consider TPC-H Q2 which - after the rewriting process - has a QEP as visualized in Figure 5.7. Note that each *Xchg* operator has an subscript ($Xchg_{m,n}$) which defines the number of producer (m) and consumer threads (n) used.

The QEP in Figure 5.7 suggests the following PPs:

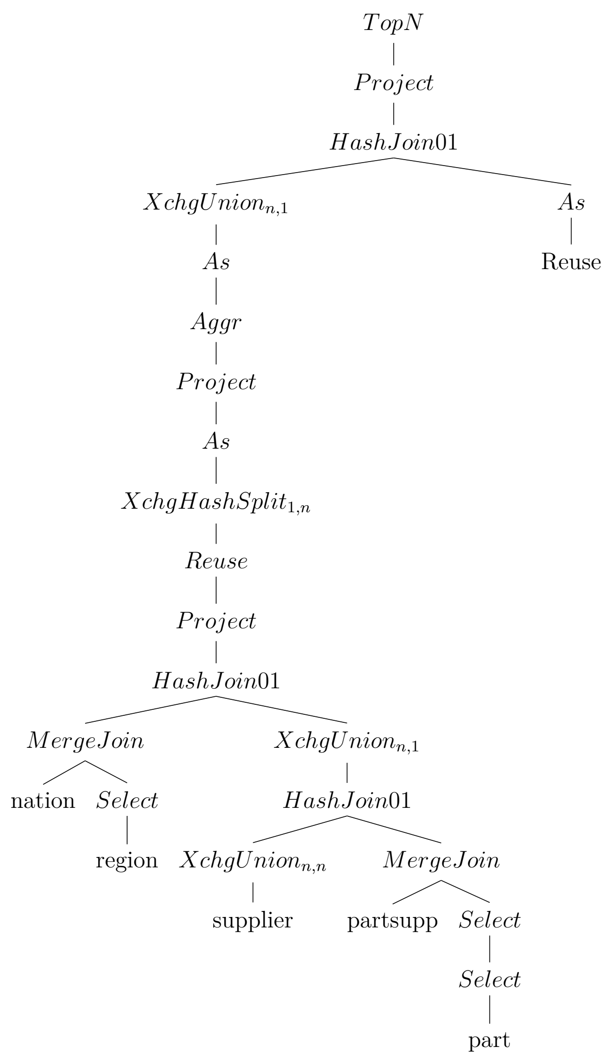


Figure 5.7.: QEP of TPC-H Q2

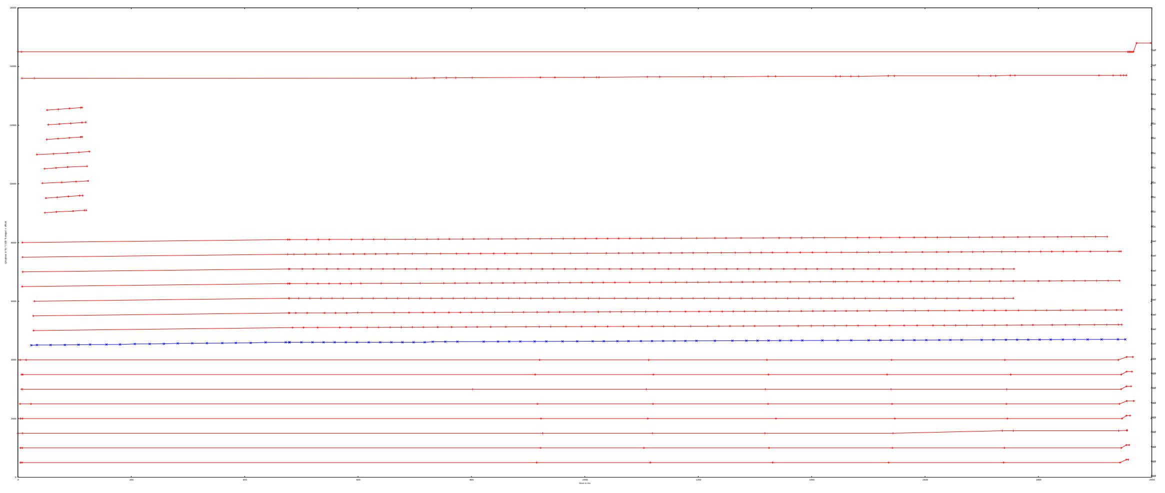


Figure 5.8.: TPC-H Q2 using 8 ULTs

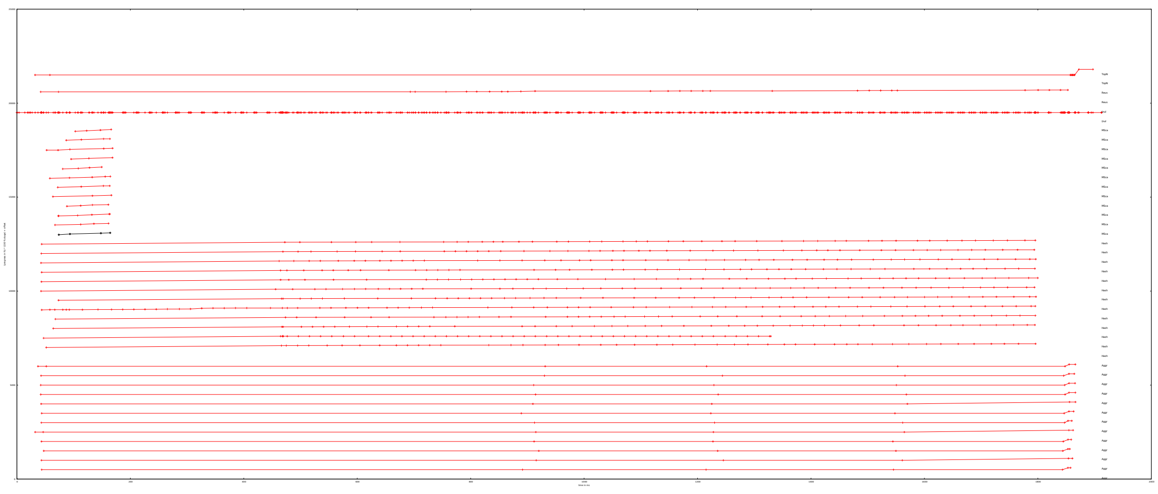


Figure 5.9.: TPC-H Q2 using 12 ULTs

1. The parallel scan of *supplier* which produces the data in order to build the VHT sequentially.
2. The parallel probing through the VHT which has been built by the first PP.
3. Above the *Reuse* operator the QEP is parallelized again in order to parallelize the *As*, *Project*, *Aggr* and another *As* operator. Afterwards the stream is sequentialized again.

In order to visualize the balancing behaviour of the implemented user-level scheduler the estimated progress of each ULT and the top-most sequential operators is plotted over the common time axis, which is the x-axis in the following plots. These plots will be used to trace the balanced achieved by the user-level scheduler.

Such a plot was created for Q2 ran using 8 ULTs. This plot is visualized in Figure 5.8. It shows that the progress reported by

- The top-most operators as the first red line (counting from the top).
- The *Reuse* operator (the one below the *XchgHashSplit* operator) as the second red line.
- Followed by eight red lines which represent the eight parallel scans which are mentioned as the first PP.
- These are followed by eight red lines that represent the eight threads probing through the shared VHT (second PP).
- And last but not least the third PP which includes the parallel *Aggr* operators.

Further it can be seen that the first PP (parallel scan of *supplier*) may be considered as balanced, because most ULTs complete their work at approximately the same time. In

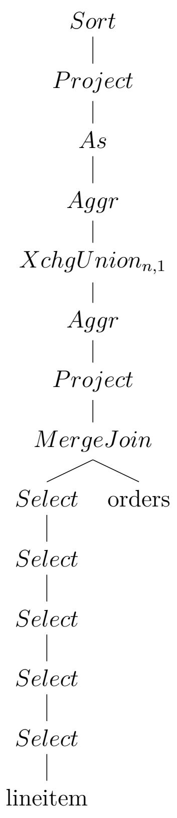


Figure 5.10.: QEP of TPC-H Q12

contrast the ULTs that probe through the VHT in parallel (second PP) are not, but still the third PP can be considered as balanced.

Now consider the plot using 12 ULTs as it is visualized in Figure 5.9. Note that one can see the same groups of threads in the run with overallocation. It can be seen that the threads the scan *supplier* in parallel (first PP) are a slightly bit more balanced than in Figure 5.8. The second PP, being the 16 ULTs that probe through the shared VHT, can also be considered as balanced, except for one ULT. The one thread completing its work early does not have a big influence on the other scheduling decisions, because there are 3 ULTs left to be used for further balancing. Also note that this behaviour happens when the time of completion is not estimated accurately due to fluctuations, which have more extreme consequences in short-running queries. Further also the ULTs that run the parallel *Aggr* operators (third PP) can be considered as balanced.

Q12: The second analyzed query is TPC-H Q12. Its QEP is visualized in Figure 5.10. It can be seen that Q12 mostly consists of parallel running *Aggr* operators (and their subtrees). In the plotted progress per ULT over the time. This will show a set of 8 lines (in the first run) or 12 lines (in the second run). Consider the plot of the first run, which is visualized in Figure 5.11. The estimated progresses over time of the mentioned parallel

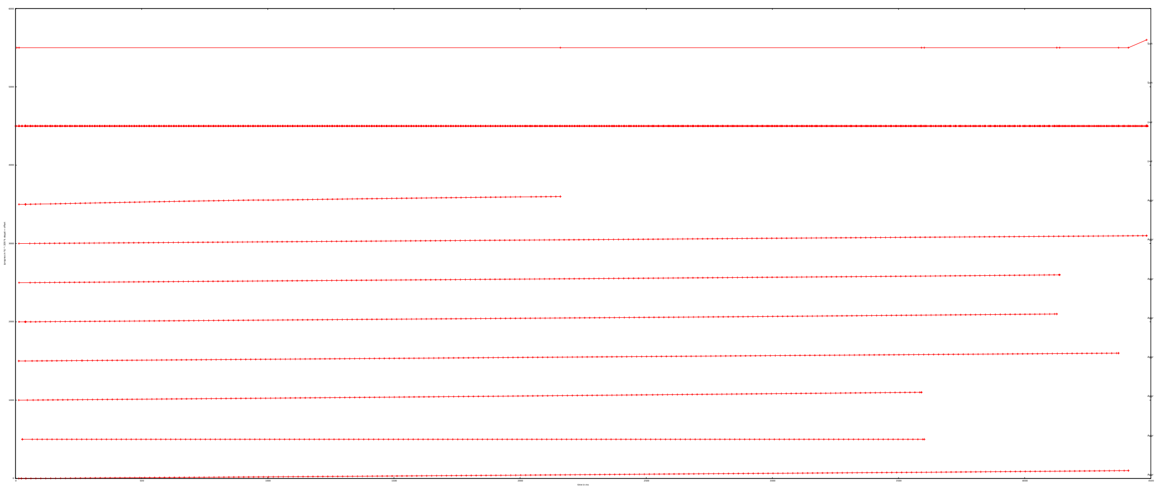


Figure 5.11.: TPC-H Q12 using 8 kernel threads

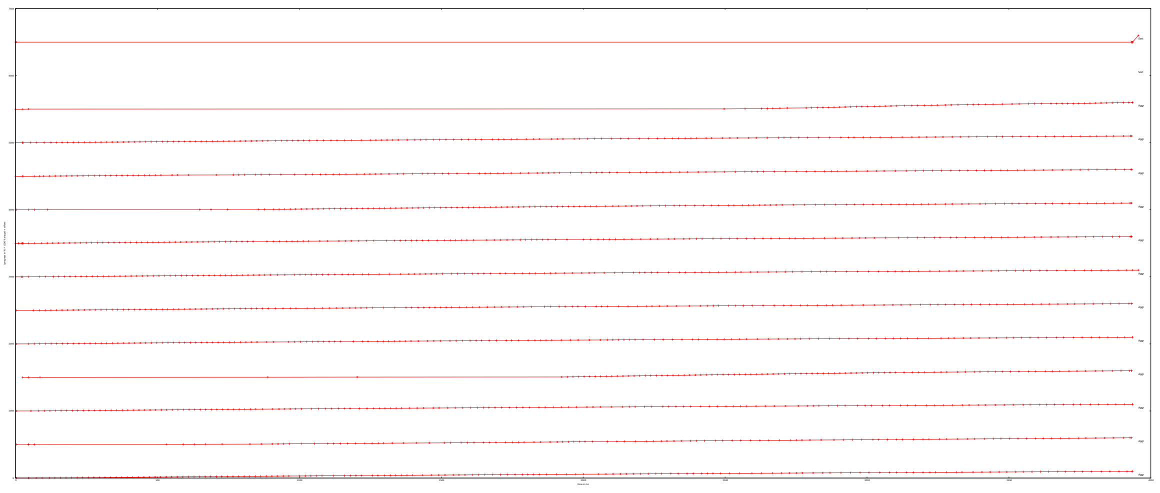


Figure 5.12.: TPC-H Q12 using 12 ULTs

running *Aggr* operators are the last 8 ones. Note that these ULTs show a noticeable imbalance, because the times each ULT ran is highly different.

Consider the plot of the second run which uses 12 ULTs as visualized in Figure 5.12. Further note that the last 12 ULTs correspond to the 8 ULTs of which each ULT runs *Aggr* operators (and its subtrees) in parallel. It can be seen that these parallel *Aggr* operators which were unbalanced in the run using 8 ULTs are balanced in this run using 12 ULTs.

5.3.2. Microbenchmark

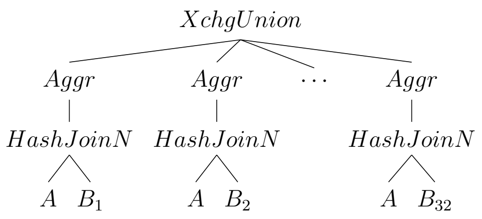


Figure 5.13.: Microbenchmark QEP

The microbenchmark uses the same query as visualized in Figure 5.13. In essence it is a parallelized hash join's with shared VHT where the VHT is built sequentially from the relation A. Further the relation B is split into 32 equal-sized partitions $(B_1, B_2, \cdots, B_{32})$ in combination different data distributions on the probe site. These different data distributions lead to a differing number of matches in the hash table and therefore a different output of the *HashJoinN* operator. Further a duplicate elemination on the top of each *HashJoinN* operator was used to amplify the execution skew.

Table 5.3.: Query response time in microbenchmark (on 32 processors)

Query	Vanilla in s	40 user-level threads in s	80 user-level threads in s
1	206.5	171.7	86.8
2	103.5	86.8	57.9
3	52.4	44	31.6
4	21.5	17.9	15.7
5	11	10.6	10.5

Response times: Running these queries on 32 processors led to the in response times as they are visualized in Figure 5.14 which shows the time spent for evaluating the microbenchmark query with differing amount of skew. It shows the query response times of five queries with a decreasing amount of data skew (from Q1 to Q5) utilizing:

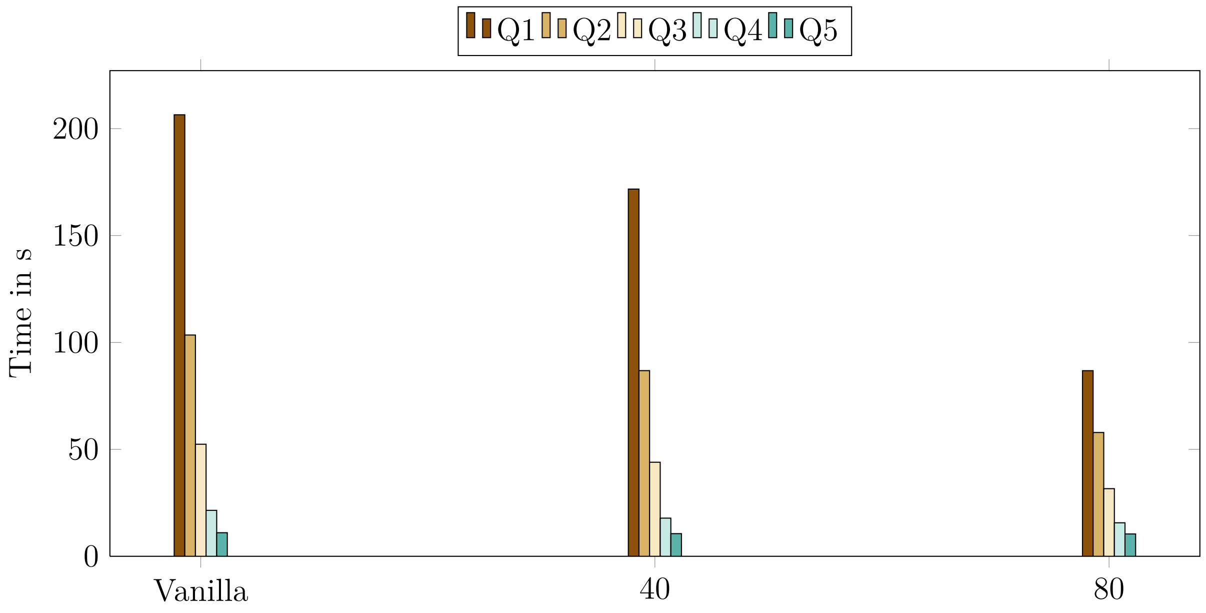


Figure 5.14.: Skew microbenchmark decreasing skew from extreme data skew in Q1 to no data skew in Q5 (on 32 processors)

- *Vanilla* Vectorwise,
- Vectorwise with load-balancing and overallocation, where the latter one uses 40 ULTs (visualized as "*40*"), and
- Load-balanced and overallocated version of Vectorwise using 80 ULTs (visualized as "*80*").

Q1 , which has an extreme amount of data skew, shows an improved query response time using load-balancing and overallocation. Utilizing 40 ULTs leads to a reduced query response time by ≈ 17% where using twice as much ULTs (80) the query response time can be approximately halfed in Q1.

Q2 has less skew than Q1, which leads to less improvement considering query response time where using 40 ULTs lead to a reduction by ≈ 16%. Using 80 ULTs the query response time of Q2 could be reduced by ≈ 44%.

Q3 shows a less improved query execution time, which can be justified by the fact that Q3 has less data skew.

Q4 shows only a slightly reduced query execution time, because it has the least skew.

Q5 has no data skew. Further it has no reasonably improved query response time in Figure 5.14. Table 5.3 shows the query response times which were plotted in Figure 5.14. It can be seen that even in the case of no data skew the load-balanced and overallocated version improves the query response time by exploting waiting times for processing another ULT.

Summarzing Figure 5.14 and Table 5.3, it can be stated that the query response time can be reduced in the skewed case. In the case of less or no data skew, the query response time could only be improved by exploiting possible wait times by scheduling another thread at that point.

Further it can be stated that in this microbenchmark the improvement, regarding the query response time, scales with the amount of skew and - of course - the number of ULTs used for overallocation.

5.3.3. TPC-H

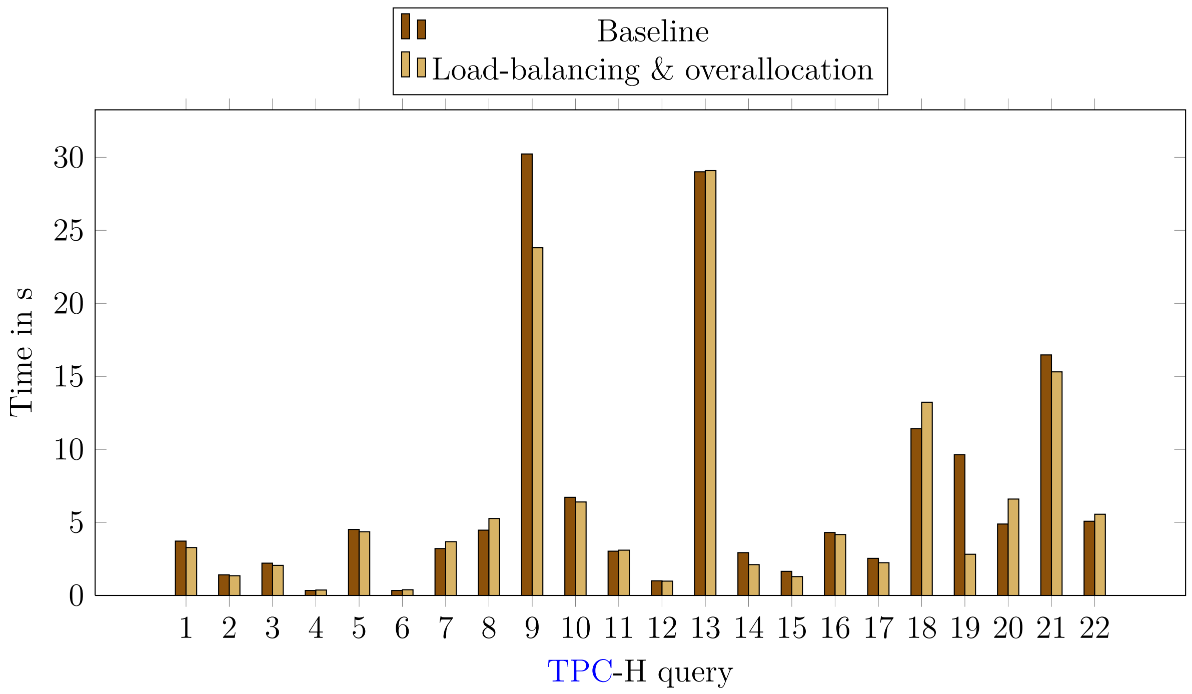


Figure 5.15.: Query response times baseline and with overallocation (on 64 processors)

The query response times for each TPC-H query are shown Figure 5.15 for vanilla Vectorwise and the load-balanced version as implemented in the scope of this thesis. Further Table 5.4 shows a subset of the profiling information gathered inside the user-level scheduler per query where for each TPC-H query (as *Query*) the number of scheduling decisions made is shown in the column *#Schedules*. The column *#Cooperative* visualizes the number of cooperative preemptions done, as well as the number of scheduling decisions caused by these. Last but not least the time spent on making scheduling decisions can be seen in the column *Overhead in cycles*. Note that this overhead does not include the overhead for context switching and the overhead that occurs due to cold caches.

Table 5.4.: Scheduler statistics

Query	#Schedules	#Cooperative	Overhead in cycles
1	$1.08 \cdot 10^4$	$1{\cdot}10^4$	$1.17 \cdot 10^8$
2	$3.97 \cdot 10^3$	$5.16 \cdot 10^2$	$1.14 \cdot 10^7$
3	$6.27 \cdot 10^3$	$1.22 \cdot 10^3$	$3.01 \cdot 10^7$
4	$1.13 \cdot 10^3$	$3.54 \cdot 10^2$	$3.28 \cdot 10^6$
5	$1.3 \cdot 10^4$	$6.77 \cdot 10^3$	$2.26 \cdot 10^8$
6	$1.23 \cdot 10^3$	$4.85 \cdot 10^2$	$6.31 \cdot 10^6$
7	$1.15 \cdot 10^4$	$5.74 \cdot 10^3$	$2.15 \cdot 10^8$
8	$2.25 \cdot 10^4$	$1.92 \cdot 10^3$	$1.15 \cdot 10^8$
9	$5.07 \cdot 10^4$	$3.41 \cdot 10^4$	$9.07 \cdot 10^8$
10	$2.33 \cdot 10^4$	$5.51 \cdot 10^3$	$1.43 \cdot 10^8$
11	$9.11 \cdot 10^3$	$9.1 \cdot 10^2$	$3.49 \cdot 10^7$
12	$3.05 \cdot 10^3$	$1.49 \cdot 10^3$	$1.52 \cdot 10^7$
13	$1.76 \cdot 10^5$	$9.36 \cdot 10^3$	$1.07 \cdot 10^9$
14	$1.53 \cdot 10^4$	$1.46 \cdot 10^3$	$1.04 \cdot 10^8$
15	$5.72 \cdot 10^3$	$1.08 \cdot 10^3$	$3.43 \cdot 10^7$
16	$1.62 \cdot 10^4$	$3.85 \cdot 10^3$	$1.14 \cdot 10^8$
17	$7.64 \cdot 10^3$	$4.85 \cdot 10^3$	$1.2 \cdot 10^8$
18	$3.51 \cdot 10^4$	$1.15 \cdot 10^4$	$1.43 \cdot 10^8$
19	$9.03 \cdot 10^3$	$6.3 \cdot 10^3$	$1.12 \cdot 10^8$
20	$1.95 \cdot 10^4$	$3.82 \cdot 10^3$	$8.14 \cdot 10^7$
21	$1.43 \cdot 10^5$	$5.56 \cdot 10^3$	$1.14 \cdot 10^9$
22	$4.14 \cdot 10^4$	$2.32 \cdot 10^3$	$1.95 \cdot 10^8$

Q1 shows a slightly improved response time using user-level scheduling and overallocation. As it can be seen from Table 5.4 the time spent for scheduling decisions was $\approx 10^8$ cycles which is mainly caused by the number of cooperative preemptions. This overhead can be reduced by enlarging the time slices, which is a trade-off between balance (smaller time slices) and performance (larger time slices).

Q2 shows almost no improvement regarding its query response time. Further it can be stated that Q2 is rather short-running where it is possible that thread setup and teardown costs hindered an improved response time. Apart from the setup/teardown costs Q2 involves a *XchgHashSplit* operator which is known to be an issue with a higher number of threads (see Section 2.2.3) compared to *Baseline* Vectorwise. In Table 5.4 it can be seen that the number of scheduling decisions made is $\approx$ 10 times the number of scheduling decisions caused by cooperative preemptions. This $\approx$ 10 times higher number is caused by the contention inside the *XchgHashSplit* operator.

Q3 - in contrast - involves no *XchgHashSplit* operators and it can be seen in Figure 5.15 that the response time of Q3 could be improved.

Q4 shows no improvement in Figure 5.15 while having one of the lowest scheduling overheads from Table 5.4. Further the - rather short-running - Q4 does not involve *XchgHashSplit* operators, but in Q4 one still has to pay rewriting and building overhead. Further ULT startup and teardown costs hindered an improved response time.

Q5 also has noticible time spent during parallelizing and building the QEP which decreased the gain that could by reached by overallocation and load-balancing. Furthermore in Q5 the scheduling overhead is $\approx$ 2 times the scheduling overhead of Q1, which also hindered an improved response time.

Q6 has a structure like Q1 (regarding its QEP). In Figure 5.15 Q6 shows no improvement. This can be explained by the rather short response time of Q6 where factors like thread setup and teardown are becoming more dominant and further avoiding hindering improvement.

Q7 shows a regressed performance as can be seen in Figure 5.15. Q7 is a query where the implemented overallocation is going to get problematic, because it increased the time of the *rewrite* phase and the *build* phase. According to the profiling information both phase took $\approx 1.5 \cdot 10^9$ cycles in sum, which is coaresly $\approx$ 1 s, whereas in vanilla Vectorwise both phases took $\approx 0.7 \cdot 10^9$ cycles. This increased time is caused by the higher number of operators to be built, because the QEP of each thread is built sequentially. Note that the load-balanced & overallocated version used 80 ULTs whereas the baseline version used 57 kernel threads.

Q8 's profiling information shows that the implemented overallocation strategy (through adapting the costs in a cost-based optimizer) did not work for this query, because the overallocated run used 12 ULTs whereas the vanilla Vectorwise could use 15 kernel threads. This results in a regressed query response time.

Q9 shows a big improvement in comparision to the other queries. According to the profiling information the vanilla version used 64 kernel threads whereas the overallocated and load-balanced version uses 80 ULTs in the most expensive part of the QEP. The profiling information showed that through dividing the work into more units i.e. ULTs it is possible to balance out extreme differing runtimes as they happen in Q9. The runtimes of the top-most kernel threads in the most expensive part of Q9 - in vanilla Vectorwise - reached from $\approx 45 \cdot 10^9$ cycles to $\approx 120 \cdot 10^9$ cycles with a mean of - coarsely $\approx 70 \cdot 19^9$ cycles. These differing runtimes could be reduced by the overallocation so that the runtimes of the ULTs (in the most expensive part of Q9) are reaching from $\approx 26 \cdot 10^9$ cycles to $75 \cdot 10^9$ cycles with a mean of $\approx 60 \cdot 10^9$ cycles. Further these more threads (with less extreme different runtimes) could then be balanced by the scheduling algorithm, which then lead to such an improved query response time.

Q10 is one of a few queries involving *XchgHashSplit* that do not show regressed response times. Further the profiling information has shown that this is another query where the rewrite- and build-time produces a noticible overhead which is $\approx 15\%$ of Q10's response time. Further from Section 2.2.3 it is known that *XchgHashSplit* operator introduce more overhead when using a higher level of parallelism. This limits the improvement.

Q11 is a query which spents a noticeable amount of the query response time in the top-most (not parallelized) operators - as known from Chapter 2. That means that no big improvement can be expected when using a virtually higher degree of parallelism (overallocation) - as can be seen in Figure 5.15. The slightly regressed response time can be explain by the higher number of ULTs for which the QEP has to be built.

Q12 shows no improvement compared to *Baseline* in Figure 5.15. The reason behind this is the lower degree of parallelism chosen by the rewriter which results in a regressed response time.

Q13 involves - according to Table 5.4 - a relatively high amount of overhead spent on making scheduling decisions. Further Q13 involves also *XchgHashSplit* operators, which are also known to scale badly. Both of these reasons avoid an improved query response time in Q13.

Q14 shows an improved query response time in Figure 5.15 while still involving *XchgHash-Split* operators. This improvement is caused by the changed cost model. Based on these changes the rewriter decided to use 24 ULTs instead of 31 kernel threads, which in combination with the *XchgHashSplit* operator resulted in an improved response time.

Q15 , which shows an improvement in Figure 5.15, also suffers from the same problem as Q14: The rewriter decided to use 16 ULTs whereas the vanilla Vectorwise used 20 kernel threads. This reduced the overhead introduced by the *XchgHashSplit* operator.

Q16 suffers from the same problem as in Q14 and Q15 where the rewriter assigned 20 ULTs whereas vanilla Vectorwise assigned 30 kernel threads to query execution. This reduced the overhead of the *XchgHashSplit* operator and lead to a slightly improved query response time.

Q17 shows an improved query response time in Figure 5.15 while involving *XchgHashSplit* operators and an overallocation of 26 additional ULTs, making 80 ULTs in sum, whereas vanilla Vectorwise used 64 kernel threads for the most expensive part of the QEP. This led to an increase build and rewriting time and higher overhead in combination with *XchgHashSplit* operators which could be amortized by load balancing the ULTs.

Q18 's response time regressed using overallocation and load-balancing. Q18 the overallocated version used 68 ULTs for scanning the relation *customer* which is afterwards sequentialized in order to build the VHT. Note that vanilla Vectorwise used 64 kernel threads for the same. After building the VHT 80 ULTs, in the overallocated version, probe through the VHT whereas 64 kernel threads were used in vanilla Vectorwise. The profiling information shows that in the overallocated version $\approx 14\%$ query time is spent on sequentializing the parallel streams before building the VHT whereas the vanilla Vectorwise spent only $\approx 4\%$ on this.

Q19 shows an improved response time in Figure 5.15. This noticeable improvement was caused by the changed QEP through the rewriter. The QEP produced by vanilla Vectorwise contains *XchgHashSplit* operators whereas the changes for the automatic overallocation caused the rewriter to choose *XchgUnion* operators instead of *XchgHashSplit* operators. This resulted in less overhead and hence an improved response time.

Q20 has a low overhead for making scheduling decisions as can be seen in Table 5.4. Despite this low overhead Q20 shows a regressed response time in Figure 5.15. The profiling information suggests that this regression is mainly caused by the extreme amount of time the rewriting process and - afterwards - building the QEP took. The sum of these times increased from $\approx 5 \cdot 10^6$ cycles to $\approx 6.7 \cdot 10^9$ cycles, in the overallocated version.

Q21 's profiling information shows that in the overallocated version less ULTs (20) were used than kernel threads in the vanilla Vectorwise (27). Further from Table 5.4 it can be seen the even with 20 ULTs has a high scheduling overhead. This high overhead is caused by the high amount of scheduling decisions which have to be made due to the *XchgHashSplit* operators. Both resulted in a regressed query response time as it can be seen in Figure 5.15.

Q22 shows a regressed response time in Figure 5.15. The profiling information shows that this regressed is caused by the *XchgHashSplit* which are becoming more expensive, when more threads are used.

Summary: All this resulted in improved TPC-H scores which can be seen in Table 5.5. It can be summarized that *Overallocated & load-balanced* version improved the *Total time* ($\sum_{i=1}^{22} t_i$ where t_i is the response time of Qi) of all as well as the *Power score* while still being able to react on dynamic workload changes.

Table 5.5.: TPC-H scores

Version	Total time in s	Power score
Vanilla	149.14	966.10 * SF
Overallocated & load-balanced	137.47	1035.82 * SF

6. Conclusion

This chapter summarizes the knowledge gained from the chapters 2, 4 and 5.

6.1. Survey

The analysis done in Chapter 2 showed the scalability of Vectorwise is limited by a set of issues.

Unscalable parallelism: One general issue in Vectorwise is the limited scalability of the *XchgHashSplit* operator, which distributes data from producer threads to consumer threads using a hash function. It was shown that the cost of involved with the *XchgHash-Split* operator increases with the number of threads being used.

Sequential fraction: Another issue is the sequential fraction that due to Amdahl's law also limits scalability. It was found out that in short-running queries with a sufficiently high degree of parallelism building the operator tree can become problematic.

Further the scalability some TPC-H queries (e.g. Q11) is limited by the top-most operators in the operator tree which are executed sequentially.

In analyzed version of Vectorwise it was not possible to build the hash table of a *HashJoin* in parallel. It was found out that this is also a major restriction for a sufficiently large inner relation.

Another example of a query part that is evaluated purely sequential is the *Reuse* operator which allows to share the results of one sub-tree (of the operator tree) with another. In case that this operator appear in the middle of a parallelized query plan, *Reuse* forces the parallel streams to be merged and afterwards parallelized again. This - not only - involves a potentially high sequential percentage, depending on the time spent inside *Reuse*. This also involve the overhead of additional synchronization, because the streams have to be merged and potentially afterwards parallelized again.

An analysis of the influence of locking on the scalability in Vectorwise has shown that locking can become a scalability killer. Analyzed was the locking inside the *Xchg* operator and inside the I/O layer. The analysis had shown that especially in *XchgHashSplit* locking is a problem, because the lock is held a relatively long time where other thread have to wait. Further the analysis had shown that the locks inside the I/O layer also have negative impact on scalability in combination of many parallel scans.

Memory locality: Due to the NUMA nature memory access can be more or less expensive depending on which data is accessed from which processor. An experiment had shown that destroying memory locality by distributing memory over all NUMA nodes had mostly positive influence on the query response time. Further it was shown that especially the access to the buffer pool and to hash tables which are shared by many threads can benefit from this. Leading to the conclusion that the performance, using the standard memory policy, was limited by the memory bandwith on a few NUMA nodes.

Skew: Chapter 2 also shows that skew between threads is problematic in the multi-core scenarios. The types of skew, that were analyzed, were caused by distribution of the data or through different processing speeds which can be due to differing memory locality or data distributions that or not partitioned into equal chunks.

6.2. Progress estimation

Chapter 4 described an scheme which estimates the progress of queries or parts of queries where the progress of an operator higher in the tree is based on the progress of the data source. In Chapter 4 these data sources included scans over existing relations and ”virtual” relations like hash tables. This led to a concept of progress that is provably monotonic increasing and provides acceptable linearity inside a stage.

One of the big drawbacks of this scheme of estimating progress is that it is non-trivial to derive a total progress over multiple stage from the progress in one stage which satisfies monotonicity and acceptable linearity.

6.3. User-level scheduler

Chapter 5 described a way to get out of the static Volcano-model parallelism by using more threads than processors and carefully balancing these threads over the processors.

In order to achieve this a user-level thread library was created. This library included the implementation of a user-level scheduler that actively tries to achieve load-balance. Further this library included the implementation of synchronization data structures (mutexes and condition variables).

Algorithm: The scheduling algorithm exploits dynamic feedback from query execution in order to achieve a balanced situation so that all threads have the same time to completion of a stage. For this the - in Chapter 4 - described progress estimation was used to calculate the time to completion of the current stage. Further the scheduling algorithm also provides a concept of NUMA awareness by prefering threads on the local NUMA node.

Performance: The efficiency of this concept was shown over a set of benchmarks. In a microbenchmark utilizing different degrees of skew showed that it was possible to reduce skew by by using a higher number of user-level threads and balancing these. Further it was possible to improve the overall time and the power score over the TPC-H benchmark.

Proof-of-concept: The implementation has shown that it is possible to loosen the hard limits set by the static Volcano-model parallelism. Exploiting overallocation makes it possible to utilize gaps produce by dynamic effects or skew for query execution. Further the dynamically scheduled approach has the benefit that computing resources can be assigned dynamically and do not have to be set statically.

Drawbacks: The implemented solution has different drawbacks which are:

- In highly concurrent scenarios the scalability issues from Chapter 2 will become more critical.
- Manual stack management and uncertainty about the stack size: a too small stack size may overflow during query execution and too large stack size may waste memory in high concurrency scenarios while still only a part of the stack is used.
- The usage of a $O(n)$ scheduling algorithm, where n is the number of threads on a NUMA node, may not be a good choice for highly concurrent scenarios. Given a sufficiently large amount of user-level threads this will perform badly. Even worse this time is spent in side a mutually execlusion section which only can be one thread at a time.
- Such an almost fully-blown user-level scheduler consumes a lot of development time. This includes especially

- that testing the correct behaviour is non-trivial and time consuming, because automatization is only possible in rare cases and
- the fact that it is hard to implement an effcient scheduler including synchronization APIs that, in certain scenarios, does not perform much worse than established and mature threading library pthreads.

7. Future work

The work that has been done in the chapters 2, 4 and 5 may be improved by in the future. This may involve a more detailed analysis in Chapter 2, a progress estimation scheme which might be able to handle more cases or an improved user-level scheduler.

7.1. Survey

The analysis provided in Chapter 2 has made assumptions that are not realistic in a general DBMS, because e.g. only hot runs on static data were considered. Furthermore it was not detailed enough in fine-grained aspects.

Cache line contention and false-sharing: In multi-core systems it can be shown that highly accessed cache lines are problematic because they involve a lot of cache invalidations and hence produce a lot of additional traffic on the interconnections. This behaviour is generally hard to analyse and therefore was omitted in Chapter 2, mainly because of missing tools for such an detailed analysis. In [ZM13] a new tool was announced this year for analyzing such issues which is an extension to Linux perf˙events.

Cold runs As explained in Section 2.4.1, Vectorwise uses specialized worker threads that load the blocks from persistent storage. Further it was assumed that the NUMA locality of the blocks read by these threads is randomized which is not realistic. Given the case that one worker thread has an affinity regarding its NUMA locality i.e. the Linux scheduler would preferably execute the thread on this NUMA node (or even processor), the assignment between blocks and NUMA could be determined more accurately.

Appends, Inserts, Updates: In Chapter 2 it was assumed that the underlying data stays constant i.e. read-only queries were considered. Appends, inserts and updates may be critical, because they involve PDTs[1] which define an delta between the stable storage

[1]Positional delta trees

and the last update(s) [HZN+10]. Hence they create another shared resource that may have a critical influence on scalability.

Disk spilling In DBMS were large amounts of data are processed operator may spill to disk according to a given policy. This - obviously - involves I/O which was avoided in the scenario analyzed. Thus taking disk spilling into account may change the analysis of the NUMA loclaity done for some operators.

7.2. Progress estimation

The in Chapter 4 has flaws that makes it sub-optimal in general DBMS use cases.

Handling I/O: The use case of a general DBMS may involve I/O which is currently not well handled in the progress estimation as presented in Chapter 2 which has to be done when blocks are loaded or written from or to persistent memory. For example operator may spill additional data to disk, this case was not considered in the scheme of progress estimation as described in Chapter 2.

Progress estimation with dynamic scan ranges: Future use of the - in Chapter 4 - presented progress estimation scheme, may include the usage of DDCC[2]. Implementation-wise this will include - so called - *Sandwich operators* which further allow to restart operators lower in the QEP [BBS12]. This cannot be handled in the, in Chapter 4 presented, QEP progress estimation scheme, because it involves another type of *Scan* that has varying ranges.

7.3. User-level scheduler

The in Chapter 5 presented user-level scheduler could be improved in order to workaround the in Section 6.3 mentioned drawbacks or to provide new functionality.

Lock-free implementation: The algorithm of the user-level scheduler could - in theory - be implemented in a lock-free manner. This would workaround locking issues that may occur in scenarios where a high number of scheduling decisions has to be made or a high number of processors is used. This may be done via lock-free priority queues [ST03],

[2]Deep dimensional co-clustering

lock-free skiplists or even hardware transactional memory which is, according to [LKN14], introduced by the Haswell microarchitecture from Intel.

Scheduling to avoid blocking: Because user-level scheduling allows the use a customized scheduling strategy it may be possible to schedule the user-level threads in a way that they have to wait very rarely on mutexes or condition variables. Such that, given that it is known where the thread could potentially wait, to preempt the user-level thread earlier (before the mutex or condition variable is reached) and schedule it again, when the scheduler detects the it is very unlikely that the thread has to wait.

Inter-query parallelism: The implemented user-level scheduler allows it to implement a strategy for handling inter-query parallelism. This could be done in a way that computation resources can be dynamically assigned to concurrently running queries based on a given quality of service model. For example queries with higher priorities might be boosted in the scheduling decision and with a longer time slice or could get more processors assigned.

Abbreviations

API	Application programming interface
CPU	Central processing unit
DBMS	Database management system
DDCC	Deep dimensional co-clustering
FIFO	First in, first out
FPU	Floating point unit
GCC	GNU C Compiler
I/O	Input / output
LRU	Last recently used
MOESI	Modified Owned Exclusive Exclusive Invalid
NUMA	Non uniform memory access
OS	Operating system
QEP	Query execution plan
PBM	Predictive buffer manager
PDT	Positional delta tree
POSIX	Portable operating system interface
PP	Parallel pipeline
SIMD	Single instruction, multiple data
SQL	Structured query language
TBB	Thread Building Blocks
TPC	Transaction performance council
ULT	User level thread
VHT	Vectorized hash table

List of Figures

List of Tables

Listings

Bibliography

[AMD13a] AMD. Amd opteron 6300 series processor quick reference guide. January 2013.

[AMD13b] AMD. Amd64 architecture programmers manual volume 2: System programming. May 2013.

[Ani10] Kamil Anikiej. Multi-core parallelization of vectorized query execution, 2010.

[BBS12] Stephan Baumann, Peter Boncz, and Kai-Uwe Sattler. Query processing of pre-partitioned data using sandwich operators. In *Workshop on Business Intelligence for the Real Time Enterprise*, Lecture Notes in Business Information Pr, page 6. BIRTE, Springer, August 2012.

[BFV99] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. Load balancing for parallel query execution on numa multiprocessors. In Athman Bouguettaya, editor, *Ontologies and Databases*, pages 99–121. Springer US, 1999.

[BZN05] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[CI12] Andrei Costea and Andrian Ionescu. Query optimization and execution in vectorwise mpp, 2012.

[CNR04] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 803–814, New York, NY, USA, 2004. ACM.

[DGT13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[Dre07] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11, 2007.

[DS05] Adam Dunkels and Oliver Schmidt. Protothreads-lightweight stackless threads in c. *SICS Research Report*, 2005.

[Dun14] Adam Dunkels. About protothreads. Website, 2014. Available online at http://dunkels.com/adam/pt/about.html; visited on July 28th 2014.

[Eng14] Ralf Engelschall. Gnu pth manual. Website, 2014. Available online at http://www.gnu.org/software/pth/pth-manual.html; visited on July 8th 2014.

[GKHC07] Boncheol Gu, Yongtae Kim, Junyoung Heo, and Yookun Cho. Shared-stack cooperative threads. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1181–1186. ACM, 2007.

[Gra94] Goetz Graefe. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):120–135, 1994.

[Gro14] The Open Group. The open group base specifications issue 7. Website, 2014. Available online at http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_cond_timedwait.html; visited on July 28th 2014.

[HM08] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[HZN$^+$10] Sándor Héman, Marcin Zukowski, Niels J Nes, Lefteris Sidirourgos, and Peter Boncz. Positional update handling in column stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 543–554. ACM, 2010.

[Inc14] Lockless Inc. Mutexes and condition variables using futexes. Website, 2014. Available online at http://locklessinc.com/articles/mutex_cv_futex/; visited on July 28th 2014.

[KDCN11] Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. A statistical approach towards robust progress estimation. *Proceedings of the VLDB Endowment*, 5(4):382–393, 2011.

[LBKN14] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. 2014.

[LFV$^+$12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[LKN14] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 580–591. IEEE, 2014.

[Mic14] Microsoft. Fibers. Website, 2014. Available online at http://msdn.microsoft.com/en-us/library/windows/desktop/ms682661%28v=vs.85%29.aspx; visited on September 1st 2014.

[MK07] Chaitanya Mishra and Nick Koudas. A lightweight online framework for query progress indicators. In *ICDE*, pages 1292–1296, 2007.

[MZZ08] Mario Milicevic, Krunoslav Zubrinic, and Ivona Zakarija. Dynamic approach to the construction of progress indicator for a long running sql queries. *international journal of computers*, (4), 2008.

[RAB$^+$13] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

[Rei07] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[SBZ12] Michal Switakowski, Peter Boncz, and Marcin Zukowski. From cooperative scans to predictive buffer management. *Proc. VLDB Endow.*, 5(12):1759–1770, August 2012.

[SC09] Xian-He Sun and Yong Chen. Reevaluating amdahls law in the multicore era. 2009.

[ST03] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 11–pp. IEEE, 2003.

[Sut09] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. Website, 2009. Available online at http://www.gotw.ca/publications/concurrency-ddj.htm; visited on March 30th 2014.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[TFB$^+$11] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. May 2011.

[Vyu14] Dmitry Vyukov. Faster fibers/coroutines. Website, 2014. Available online at http://www.1024cores.net/home/lock-free-algorithms/tricks/

fibers; visited on July 28th 2014.

[WMT08] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[ZB91] Songnian Zhou and Timothy Brecht. Processor-pool-based scheduling for large-scale numa multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 19(1):133–142, April 1991.

[ZM13] Done Zickus and Joe Mario. Numa - is it hurting your application performance? Website, 2013. Available online at http://www.redhat.com/developerexchange/DevExchange_NUMA_Performance_Debugging_Zickus_Mario.pdf; visited on July 20th 2014.

Appendices

A. Source code of the ping-pong microbenchmark

```
#include <string.h>
#include <ucontext.h>
#include <malloc.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdint.h>

#undef _FORTIFY_SOURCE

// Define when only swapcontext shall be used
// #define uctx

inline uint64_t ticks() {
  unsigned long long a, d;
  __asm__ __volatile__("rdtsc" : "=a" (a), "=d" (d));
  return ((unsigned long long)a) | (((unsigned long long)d)<<32);
}

void* allocate_stack(size_t size) {
  void* result = NULL;
  if (posix_memalign(&result, getpagesize(), size) != 0)
    return NULL;
  else
    return result;
}

size_t const stack_size = 64*1024;

/* Fiber implementation taken from http://www.1024cores.net/home/lock free
    algorithms/tricks/fibers
 * and modified */
```

```
#ifdef uctx
typedef ucontext_t fiber_t;

void wrap_call(void* fun) {
  void(*ufnc)(void) = fun;
  ufnc();
}

void create_fiber(fiber_t* fib, void(*ufnc)(void)) {
  getcontext(fib);
  fib >uc_stack.ss_sp = allocate_stack(stack_size);
  fib >uc_stack.ss_size = stack_size;
  fib >uc_link = 0;
  makecontext(fib, ufnc, 1, NULL);
}

inline void switch_to_fiber(fiber_t* fib, fiber_t* prev) {
  swapcontext(prev, fib);
}

#else
#include <setjmp.h>

typedef struct {
    ucontext_t fib;
    jmp_buf jmp;
} fiber_t;

typedef struct {
  void(* fnc)(void);
  jmp_buf* cur;
  ucontext_t* prv;
} fiber_ctx_t;

static void fiber_start_fnc(void* p) {
  fiber_ctx_t* ctx = (fiber_ctx_t*)p;
  void (*ufnc)(void) = ctx >fnc;
  if (_setjmp(*ctx >cur) == 0)
  {
    ucontext_t tmp;
    swapcontext(&tmp, ctx >prv);
  }
  ufnc();
}

void create_fiber(fiber_t* fib, void(*ufnc)(void)) {
  getcontext(&fib >fib);
  fib >fib.uc_stack.ss_sp = allocate_stack(stack_size);
  fib >fib.uc_stack.ss_size = stack_size;
  fib >fib.uc_link = 0;
  ucontext_t tmp;
  fiber_ctx_t ctx = {ufnc, &fib >jmp, &tmp};
  makecontext(&fib >fib, (void(*)())fiber_start_fnc, 1, &ctx);
  swapcontext(&tmp, &fib >fib);
}

inline void switch_to_fiber(fiber_t* fib, fiber_t* prv) {
  if (_setjmp(prv >jmp) == 0)
    _longjmp(fib >jmp, 1);
}

#endif

fiber_t ping_fiber;
fiber_t pong_fiber;
fiber_t main_fiber;

uint64_t ping2pong = 0;
uint64_t pong2ping = 0;

#define N (1024*1024)
uint64_t count = N;

uint64_t diff[N];

void ping() {
  while (count > 0) {
    ping2pong = ticks();
    switch_to_fiber(&pong_fiber, &ping_fiber);
    diff[count] = ticks()  pong2ping;
    count  ;
  }
  switch_to_fiber(&main_fiber, &ping_fiber);
}

void pong() {
  while (count > 0) {
    pong2ping = ticks();
    switch_to_fiber(&ping_fiber, &pong_fiber);
    diff[count] = ticks()  ping2pong;
    count  ;
  }
  switch_to_fiber(&main_fiber, &pong_fiber);
}

int main() {
  /* do some black magic to get the fiber for this thread */
  memset(&main_fiber, 0, sizeof(fiber_t));

  size_t i;
  for(i=0;i<N;i++) {
    diff[i] = 0;
  }

  create_fiber(&ping_fiber, &ping);
  create_fiber(&pong_fiber, &pong);

  switch_to_fiber(&ping_fiber, &main_fiber);

  /* analyze recorded stats and discard the first 1024 samples */
  uint64_t sum = 0;
  count = 0;
  uint64_t min = 42424242;
  uint64_t max = 0;
  for(i=1024; i<N; i++) {
    sum += diff[i];
    count++;
    max = diff[i] > max ? diff[i] : max;
    min = diff[i] < min ? diff[i] : min;
  }

  printf("Mean: %llu\nMin: %llu\nMax: %llu\n", sum/count, min, max);
  return 0;
}
```

Erklärung