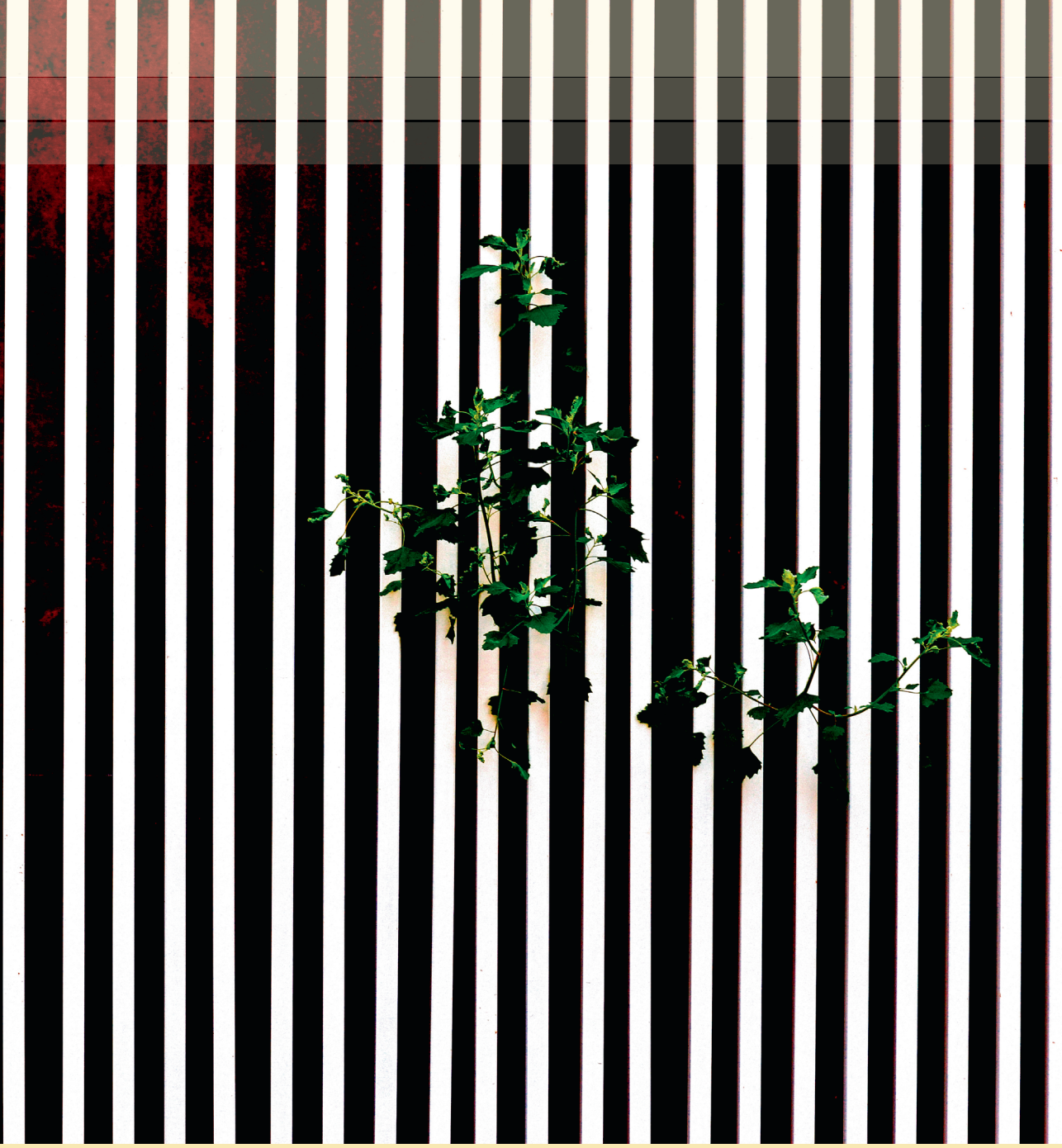

Multi-core parallelism in a column-store

Mrunal Gawade

MULTI-CORE PARALLELISM IN A COLUMN-STORE

Mrunal Gawade

MULTI-CORE PARALLELISM IN A COLUMN-STORE

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K.I.J. Maex
ten overstaan van een door het College voor
Promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op woensdag 15 februari 2017, te 11.00 uur

door Mrunal Madhukar Gawade
geboren te Belgaum Karnataka, India

Promotiecommissie

Promotor:	Prof. dr. Martin Kersten	Universiteit van Amsterdam
Copromotor:	Prof. dr. Stefan Manegold	Universiteit van Amsterdam

Overige leden:

Prof. dr. Thomas Neumann	Technische Universität München
Prof. dr. Jens Teubner	Technische Universität Dortmund
Prof. dr. Cees de Laat	Universiteit van Amsterdam
Dr. Andy D. Pimentel	Universiteit van Amsterdam
Dr. Michael Lees	Universiteit van Amsterdam

Faculteit der Natuurwetenschapen, Wiskunde en Informatica

The research reported in this thesis has been mostly carried out at CWI, the Dutch National Research Laboratory for Mathematics and Computer Science, within the Database Architectures group.

The research reported in this thesis has been mostly carried out as part of the continuous research and development of the MonetDB open-source database management system.

SIKS Dissertation Series No. 2017-04
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The research reported in this thesis has been funded through the COMMIT grant, WP-02.

ISBN: 978-94-028-0490-4

Cover designed by Mrunal Gawade (www.mrunalg.com)
Cover photograph credits - Jiang Hongsheng

Contents

Abstract

While the amount of data generated and collected keeps on growing rapidly, the added value is not the data itself but comes from finding patterns and extracting information and knowledge from the data. Analytical data management systems are the primary tools to facilitate efficient analysis of huge data volumes.

The computing power of a single machine keeps improving at an unabated rate. However, the increase of computing power no longer stems from increased clock-speed, but rather from extensive parallelization inside the CPU including multi- and many-core architectures. Consequently, (re-)designing existing and new software, including data management systems, to efficiently and effectively using the entire available computing power of rapidly changing and ever more complex hardware architectures has become an important science and engineering challenge.

While analytical database query processing in principle lends itself well to parallelization, finding a good (let alone optimal) way of parallelizing an arbitrary given query depends on numerous parameters including the kind and complexity of the query itself, detailed characteristics of the ever more complex hardware architectures, characteristics of the data and concurrent workloads (some of which, in particular the latter two, might not be known upfront), and is known to be a computationally hard problem.

The research reported in this thesis addresses several challenges of improving the efficiency and effectiveness of parallel processing of analytical database queries on modern multi- and many-core systems, using an open-source column-oriented analytical database management system, MonetDB, for validation. In contrast to the existing work we also broaden the research from focusing on individual operators and algorithms to consider the entire system and process holistically.

A prime prerequisite to achieving resource-efficient parallel query execution is a detailed understanding of the impact of the various parameters sketched above. Recognizing limitations of existing techniques and tools, we design and develop new visual analysis techniques and tools that help to identify and rank performance bottlenecks of parallel query execution on multi-core systems.

Deploying these tools in multiple showcases revealed that in particular as the number of CPU cores grow rapidly with multi- (let alone many-) core CPUs, finding an optimal degree of parallelization becomes increasingly difficult. Static parallelization techniques easily fail by using too low degree of parallelism, and thus leaving resources (cores) unused, or using too high degree of parallelism, and thus suffering from synchronization and other overheads.

This observation inspired us to design and develop a novel learning based adaptive technique for multi-core parallel plan generation using query execution feedback. This techniques proves to be particularly efficient with concurrent workloads, a scenario which is very common in practice but has been largely uncharted in database query parallelization research.

To further increase the compute power of a single machine, multi-socket systems (accommodating multiple multi-core CPUs) have become a commodity. However, while providing transparent access from each CPU to the entire available memory, access performance is non-uniform, i.e., each CPU has faster access to ”its

own" local part of the memory, but slower access to the "remote" memory of other CPUs. Ignoring this non-uniformity in memory access in parallel database query evaluation leads to non-optimal performance and undesired performance variations.

We show that using a simple technique where a multi-socket system is treated as a distributed shared nothing database system, the remote memory accesses could be constrained thereby having a controlled query execution performance.

Many-core system architectures are the latest trend to imitate GPU style parallel execution where there are 240 threads on 60 cores in a Xeon-Phi processor. However, data transfer on the PCIe bus which connects Xeon-Phi co-processor to the host, is a bottleneck due to the limited bandwidth. We analyzed the effect of streaming execution of selected queries, to utilize PCIe bandwidth optimally, to understand possibility of Xeon-Phi knights corner architecture usability in data analytical workloads.

This thesis contributes to our understanding of the multi- and many-core CPU landscape in the context of analytical database systems, exemplified by the MonetDB columnar system. Many of the lessons, experiences and insights gained are valuable for the emerging analytical database systems.

Samenvatting

De hoeveelheid gegenereerde data blijft snel groeien, het vinden van betekenisvolle patronen in zulke data is cruciaal. Analytische database systemen zijn een van de belangrijkste hulpmiddelen om dit te bereiken.

De rekenkracht van computers neemt nog steeds snel toe. Niet door het verhogen van de kloksnelheid, maar door het gebruik van parallelle verwerking middels multi- en many-core systemen. Het her-ontwerp van bestaande en nieuwe software, waaronder database management systemen, om hier optimaal gebruik van te kunnen maken is een belangrijke wetenschappelijke uitdaging.

Analytische database systemen gebruiken query parallellisatie als een standaard techniek om de verwerkingstijd van analytische queries te verbeteren. Query parallellisatie is een wetenschappelijke uitdaging dat op een holistische manier moet worden benaderd. Een van de eerste stappen is identificatie van performance bottlenecks tijdens het uitvoeren van queries. Goede visualisatie technieken zijn hierbij cruciaal om zulke bottlenecks snel te vinden. We hebben zo'n visualisatie techniek ontwikkeld en daarmee het gedrag van de parallelle verwerking van queries op MonetDB onderzocht.

Een van de belangrijkste factoren is het vinden van de optimale graad van parallellisme. Query plannen worden normaal gesproken gegenereerd zonder variaties in runtime mee te nemen, waardoor de performance gedurende het parallel uitvoeren van queries lijdt onder de variabiliteit van runtime resources zoals CPU, memory, etc. Een statische parallellisatie techniek kiest vaak een te laag nivo, waarbij cores ongebruikt worden, of leidt tot een te hoge parallelisatie graad met als gevolg congestie op de toegang van cruciale hulpmiddelen. Deze observaties hebben ons aangezet tot het introduceren een nieuwe techniek voor het genereren van mulitcore parallelle executie plannen op basis van Machine Learning technieken, die gebruik maakt van execution feedback.

Om de rekenkracht verder te verhogen zijn multi-socket machines geintroduceerd. Ogenschijnlijk bieden ze de programmeur transparente toegang naar elk stuk van het geheugen, maar is de performance sterk afhanklijk van toegang tot local of remote geheugen. Dit lijdt tot problemen met niet-uniforme memory access en variatie in de prestaties van query verwerking. We hebben laten zien dat met een simpele techniek, waarin een multi-socket systeem wordt behandeld als een gedistribueerd database systeem, de remote memory acces kan worden beperkt met als gevolg gecontroleerde query executie prestaties.

Many-core systeem architecturen zijn de laatste trend die probeert een GPU-achtige parallelle executie te realiseren. Een voorbeeld is de Xeon-Phi processor met 240 threads op 60 cores. Data transfer over de PCIe bus die de Xeon-Phi coprocessor verbind met de host is echter een bottleneck door de beperkte bandbreedte. Om optimaal gebruik te kunnen maken van de beschikbare PCIe bandbreedte hebben we het effect van streaming executie van een aantal geselecteerde queries geanalyseerd en om te begrijpen of het mogelijk is om de Xeon-Phi knights corner architectuur te gebruiken bij data-analytische workloads.

Met dit proefschrift hebben we een bijdrage geleverd aan het beter uitnutten van multi- en many-core CPUs in de context van een analytisch database systeem, geil-

lustreerd aan de hand van het systeem MonetDB. De lessen, ervaringen en inzichten zijn van groot belang voor de verdere ontwikkelingen in dit veld.

Acknowledgments

My sincere thanks to my adviser Professor Martin Kersten who offered me the opportunity to do research in the database group at CWI. Thank you Martin for listening to all my complains, and all other forms of concerns that a student on the PhD course goes through in the starting period. I am ever grateful to you for your caring attitude in those troublesome days. Thank you for all the boot-up codes that you initiated to get me accelerated on MonetDB code base development. It was very useful to have such hand holding in the initial days, to keep myself going.

My next big thanks to my co-adviser Professor Stefan Manegold. Stefan would help resolve any doubts with meticulous details, where we would end up having extended discussions. These discussions with you were always insightful. You were always a kind boss, who offered support during all administrative hassles, with detailed guidelines in case of any doubts. Your attention to details attitude helped me to inculcate those qualities.

Niels, a lot of thanks to you for tolerating my constant interruptions when I needed you. Discussions with you were always fun and I learned a lot from you. Thanks to you and Arjen for all the system administration related query resolutions at odd hours using emails.

Sjoerd, many thanks to you for all the help in getting any MonetDB related issue resolved when I needed. Thanks for all the document corrections and English corrections in the paper drafts. I learned new things in English from you.

Thanks Peter for your critical comments during various group presentations. They really helped me to prepare during research presentations.

Aram, my ex-room mate taught me lot of things about the European culture and was a big support during many a crisis situations. I miss you and our deep conversations always. It was great to share house with you and learn so many things from you. I wish you all the good luck in your career.

Thanks to Ashwin Rao, my long time friend who recommended MonetDB group and who helped me in different ways during PhD.

Database architecture group is like a family. Thanks to all the seniors who always supported all my questions and offered help and advise when needed. Thank you Stratos for all the inspirational talks in my early days as a PhD student. Your advise was always very useful. Erietta thank you for all the help in getting used to the cultural aspects. It was very useful always. Lefteris thank you for your helpful talks during the early years, when I used to feel lost and during difficult times. They helped tremendously. Romulo, thanks for all the technical advise in the initial days and offering the friendly personal advise when I needed it.

Eleni, you are a true friend, whom I contacted anytime I needed a friendly hand, and a shoulder to cry on. Your entire family is wonderful and very kind. Thank you for being there to give me a homely feeling when I desperately needed it.

Duc, thank you for all the help that you provided during my last year. Thank you for listening to all my wild ideas and brainstorming them with me. It was great to have you around and long discussions that we had on various topics. Thank you so much. Yagiz, my first year office mate, thank you for your philosophical views on various topics when we discussed them.

Hannes, thank you for motivating me during paper review process and all the insightful discussions we had on my papers and otherwise. It is always great to have you around and your technical energy in various fun projects is a source of constant inspiration. I wish you a great success in the research world.

Holger, thank you for all the conversations we had and motivating me with your energy during group discussions. It was great to have you in the group and you are surely being missed. Thank you for referring me for the Oracle Labs internship.

Thibault, I so wished I could also go on a music concert tour like you. Luckily I had my own small crowd-funding project to get a feel of artistic work :-). It was always nice to hear different aspects of the European culture from you. Jennie, Dimiter and Robin. Thank you for all the friendly discussions on various topics. Meng it was nice to have you as an office-mate. I wish you good luck in your future plans. Kostis and Photeini, I learned a lot about Greek culture from you both. Thanks to Sandor, Bart, Fabian for helping me with different discussions when needed. Thank you Martine, Minnie, Irma, for all the administrative help. Myriam thank you for all your help in the initial period.

Thanks to Chris Wesseling for interesting discussions and help on new ideas on my new initiative ScienceParkCreatives towards end of my PhD. Thanks also to Pedro and Abe the Masters interns in my office for interesting discussions.

Ashutosh was the only Indian in CWI that I would converse some times. Thanks Ashutosh and good luck for PhD. Domenico my Italian room mate for 1 month offered warm friendship. Thank you.

Many thanks to Adel, my residential building's care taker who offered me coffee and friendly advise about European culture. My Dutch house mates in DUWO housing complex Daniel and Mirte for extending their warm friendship. Thanks to Gerrit and Cindy for being good Dutch friends and extending their warmth when I did not know anybody in Amsterdam. My thanks to also the Boer family who hosted dinners to make us feel integrated in the culture. Many thanks to the Park family who took good care of me during my Oracle Labs internship in California. Thanks to Raoul, the owner of the cafe Maslow to offer legal advise when I needed it on different matters.

Thanks to Reindert Hommes for helping in opening up ScienceParkCreatives innovation lab and all the support. Thanks Chantelle Pickee and Michiel Witlox for your faith in me as the Brand Ambassador of Science Park Amsterdam.

Finally, a big thanks to my family and old distant friends everywhere in the world, for providing all the support needed to stay strong during many crisis moments. You are my hero.

Chapter 1

Introduction

"Information is not knowledge." – Albert Einstein

1.1 Data is everywhere

The amount of data generated keeps on growing each year. Most of this data is generated through web related activities such as automated logs, instant messages, mail communication, media sharing in terms of photo, audio, video, etc. A large amount of data is also generated through scientific experiments in the fields of astronomy, seismology, particle accelerators and so on. The latest addition to the data generation sources is through the Internet of things and wearable devices.

The data in itself is useless unless we can analyze it to find meaningful patterns to extract knowledge. Hence, data analytics, also known as data science, is one of the hottest research area today. The data analytics solution preference is largely driven by the size and the volatility of the data in use. Statistical software packages such as *R* are commonly used by data scientists for dealing with small data sets that fit in a single machine's memory [54]. Many data scientists also use Microsoft Excel based spread-sheet solutions for small data sets [83]. Map-reduce based data analytics solutions come into play for dealing with large peta-byte scale data sets usually employed by Internet based companies [57].

Analytical database systems are one of the most critical pieces of infrastructure in the technology stack. With growing memory sizes many data sets can be fitted in the memory of a single powerful machine [31]. These database systems support in-memory data processing, while leveraging decades of research in data management. Unlike the file based storage supported data analytics solutions such as *R*, in analytical database systems, data can be processed by applying different data processing techniques. Furthermore systems use the widely popular SQL based front-ends for defining analytical queries, which makes writing analytic queries an easier job. Technology that helps to accelerate the process of database system analytics, is thus immensely valuable.

1.2 Data management in the hardware world

Data management using database systems is a software based solution. However, software can never exist in isolation, and works in tandem with the hardware technologies to assist in processing, storage, and transfer of data between different machines. Innovation in software thus co-exists with emerging hardware technologies. New hardware technologies such as multi-core CPUs, flash memory storage, NVRAM based storage, fast interconnects, are some examples of development in the last decade, calling for an experimental driven science to access the old and the new technologies. These developments lead to some important research questions as follows.

Question 1: How well are the state-of-the-art database management system solutions exploiting the available hardware resources?

The workloads that tackle data analytics problems are termed as the data analytical workloads. One of the prime concerns during analytical workload processing is to improve the query execution performance. Memory optimized analytical database systems are designed with analytical query processing as their focus. As they work on in-memory data, they are critically dependent on memory bandwidth and CPU processing power. CPUs themselves have a wide range of architectures that comprises of different type of caches, vector registers (SIMD), micro-architectures, interconnects, etc., which affects how data is processed and in turn affects the query execution performance. Optimized query processing thus has to take into account dependency on the underlying hardware architectures. Designing hardware aware query processing optimizations for faster query execution, using improved CPU processing and memory bandwidth are in high demand as exemplified by the tier 1 conference proceedings such as VLDB, ACM SIGMOD, DAMON etc.

Question 2: How to leverage multi-core systems to improve the performance of analytical workloads?

In a multi-core CPU each core acts as an independent CPU. Multi-socket systems use multi-core CPUs in each socket connected through fast interconnect. Now-days most multi-socket systems have two or four sockets.

Multi-core CPUs and multi-socket systems offer a many-fold of increased processing power. How to utilize this processing power most efficiently for analytical workloads is the prime focus of this thesis. Analytical systems use query parallelization as one of the standard techniques to efficiently utilize multi-core CPUs. The research reported in this thesis uses parallel query execution as the fundamental building block. A large body of research work targets exploration of query execution performance improvement in non uniform memory access based systems. We provide an overview of this work in Chapter 2.

Question 3: How does the multi-core hardware affects the query optimizers?

Query optimization is a fundamental research problem in database systems [88, 46, 86]. Optimized serial plan generation using heuristics or cost models has been extensively researched. Some prominent techniques are mid-query re-optimization [92], learning based optimization[138], and multi-query optimization [127]. Query optimization for parallel databases in a distributed system setting has been also around for a considerable time [29, 141, 117]. In a distributed system setting the network interconnect between machines play a crucial role, as it affects the bandwidth and the latency of the data transfer. Multi-core CPUs with multiple cores on a single chip are like a distributed system, however, with different architectural properties in terms of its memory hierarchy, interconnects, etc. They bring a new perspective in query optimization, as plan generation has to take into account multiple CPU cores and their architectural properties as well. In this thesis we focus on a new query optimization technique we developed for parallel plan generation for multi-core CPUs.

Question 4: How to provide insights into the query execution performance bottlenecks at a database system's functionality level?

Identifying query execution bottlenecks is of critical importance to improve performance. Hence, suitable debugging tools are of invaluable importance. Most of the tools are designed keeping system specific functionalities in mind. However, the generic principles they employ are applicable for a wide range of systems. We elaborate on different use cases of improving query execution performance using such tools.

We have highlighted the generic role of multi-core CPUs in query execution performance improvement so far. Next we briefly summarize a more focused view of the problems that arise during performance improvement using query parallelization.

1.3 Query parallelization

Query parallelization is a standard technique for improving query execution performance using multi-core CPUs. At a holistic level query parallelization involves each CPU core executing a part of the query plan and aggregating results of individual query plan execution. Query parallelization is a hard problem as it is closely related to the query execution strategy, which gets influenced by the CPU architecture in use. Some of the prominent problems in parallelized query execution are as follows.

- Identification of the optimal degree of parallelism of a query plan, for CPU architectures with a large number of CPU cores.

- Generation of an optimal parallel plan using query optimization techniques without considering the concurrent workload.
- Generation of an optimal parallel plan during concurrent workload execution, taking into account the resource contention.
- Identification of execution performance bottlenecks during parallelized query execution.
- The effect of non uniform memory accesses (NUMA) in multi-socket systems, on parallelized query execution performance.

This thesis explores the above problems in the context of the research questions raised in Section 1.2, in a multi-core CPU environment. Next we provide a summarized overview of our contributions in this context.

1.4 Contributions and covered publications

Most research in multi-core query parallelization is focused on individual aspects such as the effect of multi-core CPUs on the design and implementation of relational algebra operators and comparisons of these implementations from a speedup perspective on multi-core systems. For example, the join operator is the most researched operator in the literature [36]. However, a standalone operator performs differently when executed in a full-fledged database execution engine, due to variations in the run-time resources such as the CPU cores and memory bandwidth used by the other operators under execution. Hence, an important research question to ask is what is the holistic effect of operator based optimizations in a database system.

This thesis focuses on the exploration of the query parallelization problem in a holistic sense in an established full-fledged column-store database system. The exploration focuses on topics such as 1) visualization tools to identify performance issues during query execution, 2) a new parallelization technique for improved multi-core utilization, 3) detailed analysis of the new technique in isolated and concurrent workload environments while comparing with state-of-the-art systems, and 4) insights into utilization of new hardware characteristics such as NUMA aware execution and many-core architecture execution.

We summarize the science question researched, the research methodology used, and the corresponding publications list next.

- **Visualization tools to identify performance bottlenecks.**
  Addresses research questions 1 and 4: Identifying execution performance bottlenecks is of importance to improve parallelized query execution performance. Insights at the functional level i.e. relational algebra level are needed. Most database execution engines provide text-based analysis tools, which work reasonably well for analyzing serial query execution plans. However, parallelized query plans tend to be more complex. Other generic low level

tools such as Vtune analyzer [126] that highlight CPU load, memory bandwidth, IO etc. are insufficient to showcase database system specific requirements. Hence, any assistance that expedites the process of bottleneck identification in parallelized query execution is a crucial step in the research exploration. Operator dependency in a query plan can be represented by a graph based dependency order and a time based dependency order during execution. We explore how these two representations could be visualized to assist in query execution performance bottlenecks identification.

The research methodology: Visualization tools are better, compared to text based tools as they expedite the process of performance bottleneck identification. They quantify the information for an easy analysis using different schemes such as color coding, clustering, shapes and sizes, interactive navigation, etc. Operator's execution flow can be visualized using graph based dependency order or time based dependency order. The state of an operator's execution can be color coded to indicate the execution flow in an online and offline manner. Online analysis gives an immediate understanding of where the bottlenecks occur by a visual inspection using the color coding scheme. More detailed insights per operator such as the execution time, resource consumption, etc. can be obtained using an offline analysis. As parallel plans tend to be more complex than serial plans, availability of such tools influences the research progress.

Important performance issues such as operator scheduling problems, resource consumption problems which are impossible to identify using text-based tools can be identified using such tools. The insights obtained can lead to crucial changes in the system architecture components such as the operator scheduler, the interpreter, different operator's implementation, etc. This inspires development of similar tools suitable for other systems.

Gawade, Mrunal, and Martin Kersten. "Stethoscope: a platform for interactive visual analysis of query execution plans." Proceedings of the ***VLDB*** *Endowment 5.12 (2012): 1926-1929.*

Gawade, Mrunal, and Martin Kersten. "Tomograph: Highlighting query parallelism in a multi-core system." Proceedings of the Sixth International Workshop on Testing Database Systems ***(DBTest)****. ACM, 2013.*

- **Adaptive query parallelization.**
  Addresses research questions 2 and 3: Identifying an optimal degree of parallelization and an optimal parallel plan is a NP-hard problem. Most systems follow the *exchange* [68] operator based plan parallelization, using heuristic or cost model based approaches. Simple heuristics such as allocating all cores to the parallelized query does not result in improved performance. It could also lead to degradation due to overheads in management of the extra resources and resource contention. The cost model based parallel plans are very sensitive to operator cardinalities, hence do not generate optimal parallel plans either. Identifying an optimal parallel plan is thus an open science problem.

The research methodology: A feedback based technique to identify the optimal degree of parallelization and multi-core utilization in a parallelized query execution environment therefore is a good research direction. It is based on the methodology of learning from the past and is inspired by the observation that most analytical queries use templates. When the same query albeit with different parameters is fired, adaptive parallelization comes into effect. During each query invocation the most expensive operator in the query plan is incrementally parallelized, until an improved parallel plan is identified. A convergence algorithm stops the feedback loop when the most suitable parallel plan is identified. Adaptive parallelization improves multi-core utilization compared to heuristic parallelization. It is a result of less resource consumption as the number of data partitions in adaptive parallelization are less compared to heuristic parallelization. As adaptive parallelization uses the *exchange* operator [68] based approach, it can also be used in other database systems that use the exchange operator based parallelization.

*Gawade, Mrunal, and Martin Kersten. "Adaptive query parallelization in multi-core column stores." Proceedings of the **EDBT** 2016.*

- **Concurrent workloads and query parallelization.**
  Addresses research questions 3 and 4: Most database systems generate a parallelized query plan without taking into account run-time resource contention due to a concurrent workload. Modeling run-time resource variations is practically impossible. Investigating the effect of a concurrent workload on a parallelized query execution is crucial to make progress in creating resource contention aware parallel plans. Different types of concurrent workloads generate different types of resource contention. Also the effect of resource contention on a parallelized query execution varies significantly depending on the query parallelization technique (intra-query / inter-query) under use. Hence, investigating query parallelization under concurrent workload is a critical problem in the query parallelization landscape.

  The research methodology: A proven technique in the database research is to design different types of workloads that create different levels of resource contention. The effects of these concurrent workloads on individual parallelized query execution are analyzed using different query parallelization techniques in the context of different database systems. It provides detailed insights from thread variations, scheduling overheads, robustness of individual operator's, and intra-query against inter-query parallelization perspective. Insights are also obtained by quantifying the performance effects in terms of microarchitecture hardware counters such as cache misses, pipeline stalls, etc. Analyzing resource contention due to concurrent workload is challenging due to experimental setup complications, workload variations, ability to isolate individualized query performance effects, etc. Such type of research exploration in the context of analytical database systems for different parallelization techniques does not exist so far. The insights obtained can be used to create synthetic workloads that use intra-query or inter-query paralleliza-

tion techniques and to understand the relative merits of different parallelization techniques under resource contention.

*Gawade, Mrunal, and Martin Kersten. "Multi-core column stores under concurrent queries."Proceedings of the Data Management on Modern Hardware **DaMoN**(2016).*

- **NUMA effects on the memory mapped storage.**
  Addresses research questions 1 and 2: NUMA systems pose a particular challenge to database engines as the memory access latency and bandwidth varies based on the location of the data access. Most server-class systems use 4 socket NUMA systems. The predominant approach taken to mitigate the NUMA problem is to build new database systems from scratch, making the individual database operators NUMA aware. However, rewriting the code base of existing database engines to make them NUMA aware is a lot of effort due to the legacy code-base. Hence, solutions that make existing database engines NUMA aware are well sought after.

  The research methodology: A promising research direction is to mitigate the data affinity to memory banks problem, by letting the operating system do the scheduling using the memory mapping feature. This research direction can be explored further to analyze the NUMA effects on a memory mapped storage system during parallelized query execution.

  As remote memory accesses are the main culprit in NUMA performance, minimizing them is crucial. Hence, a new distributed system based (shared nothing) architecture is a good research direction. The data is horizontally partitioned on each memory bank and each socket is affined with a database engine execution instance (slaves), while a master database execution instance coordinates the distribution to slaves. Though such an approach has been explored in the context of transactional database systems [131], it is novel for the analytical systems. This is a simple architecture, which does not require major architecture level changes, hence legacy database systems can use it.

  *Gawade, Mrunal, and Martin Kersten. "NUMA obliviousness through memory mapping." Proceedings of the Data Management on Modern Hardware **DaMoN**(2015).*

- **Xeon Phi accelerated database.**
  Addresses the research questions 1: Use of new hardware such as GPUs and Intel Xeon-Phi many-core processors as database accelerator is an active research area. GPUs are programmed using CUDA and OpenCL programming frameworks which are relatively new, compared to Xeon-Phi's X86 based programming framework. Both GPUs and Xeon-Phis are used as a co-processor attached to the PCIe bus and are primarily targeted towards the high performance computing workloads, due to their high computing power. Their memory bandwidth is very high (upto 500 GB/sec), however, they have a limited device memory (upto 16 GB).

During database query execution data has to be either stored in the device memory or transferred over the PCIe bus to the device. However, as database workloads handle large data sizes, the limited device memory (up-to 16 GB) and the limited PCIe bus bandwidth (upto 6 GB/sec, when compared with the memory bandwidth), limit usage of these co-processor devices in database workloads. Optimizing data transfer over the PCIe bus is thus an important problem in the context of database workloads. The role of GPUs to accelerate database query execution and optimized data transfer over the PCIe bus has been explored quite well [120]. On the other hand the research exploring Xeon-Phis as a database query execution accelerator is quite primitive.

The research methodology: We propose a many-core architecture based (Intel Xeon-Phi) execution engine for database query acceleration, that gives an outlook on the next wave of innovation in many-core systems. We investigate the PCIe bottleneck for data transfer to Xeon-Phi. A new streaming based multi-threaded MPI based implementation of database execution engine to accelerate a select operator on Xeon-Phi is proposed. Xeon-Phi prices have dropped from thousand dollars to a hundred dollar, which intrigues us to investigate their role in cluster based configuration, where more than one Xeon-Phi is present in a single system. The MPI based implementation we investigate is an exploration attempt in that direction.

1.5 Thesis overview

In Chapter 2 we provide a brief background on relational database management systems, different multi-core CPU architectures, standard software parallelization system libraries such as pthreads, and latest research overview of multi-core CPUs in the context of database systems.

Many readers might be familiar with the background material already. The overview of the state-of-the-art research towards the end of the Chapter might be of specific interest to such readers.

Chapters 3 and 4 describe the visualization tools to identify query execution bottlenecks in the parallel world. In Chapter 3's Appendix we provide sample query data flow graphs to help readers get an intuition of the complexity of the parallel plans for the database system in use. In Chapter 4 we provide a detailed background analysis of some sample analytical queries, with operator level details. In both Chapters we provide different use cases to illustrate how visualization offers quick help in identifying and analyzing performance problems during parallelized query execution. These Chapters are based on the following peer reviewed publications.

*Gawade, Mrunal, and Martin Kersten. "Stethoscope: a platform for interactive visual analysis of query execution plans." Proceedings of the **VLDB** Endowment 5.12 (2012): 1926-1929.*

*Gawade, Mrunal, and Martin Kersten. "Tomograph: Highlighting query parallelism in a multi-core system." Proceedings of the Sixth International Workshop on Testing Database Systems **(DBTest)**. ACM, 2013.*

In Chapter 5 we introduce a new feedback based parallelization technique, called *adaptive parallelization*. In the first half of the Chapter we describe the new parallelization technique along with a new convergence algorithm that identifies an optimal parallel plan in minimal convergence runs. In the second half of the Chapter, we provide an extensive experimentation comparing the new technique with an existing technique under different experimental settings. This Chapter is based on the following peer reviewed publication.

*Gawade, Mrunal, and Martin Kersten. "Adaptive query parallelization in multi-core column stores." Proceedings of the **EDBT** 2016.*

In Chapter 6 we explore the effects of different types of concurrent workloads on parallelized execution of a single query. We offer detailed operator level analysis of some queries, along with summary notes, which users might find helpful for an overview. We evaluate the resource contention effects of different concurrent workloads on parallelized query execution using static, cost model, and adaptive parallelization techniques, in the context of three different database systems. The experiments address four independent questions, which readers are free to navigate independently. This Chapter is based on the following peer reviewed publication.

*Gawade, Mrunal, and Martin Kersten. "Multi-core column stores under concurrent queries."Proceedings of the Data Management on Modern Hardware **DaMoN**(2016).*

The first half of the Chapter 7 explores the NUMA effects on parallelized query execution of a memory mapped storage system (NUMA obliviousness), while the second half compares the execution performance of NUMA oblivious architecture with a new shared nothing architecture we propose, where we treat the multi-socket system as a distributed database system. This Chapter is based on the following peer reviewed publication.

*Gawade, Mrunal, and Martin Kersten. "NUMA obliviousness through memory mapping." Proceedings of the Data Management on Modern Hardware **DaMoN**(2015).*

How to use many-core architectures such as Xeon-phi in the database workloads is an important problem for the future. Towards the end we try to categorize such use cases.

The experimental approach used in this thesis led to many improvements in the MonetDB system and provides a reference for other database systems. Research in the hardware / software co-design from the database world perspective is thus a never ending story.

Chapter 2

Relational databases and multi-core CPU landscape

"Individually, we are one drop. Together, we are an ocean." — Ryunosuke Satoro

In the first half of this Chapter we provide the background material on relational databases to help understand their role in data management solutions. We cover the landscape from the database system components, different types of workloads, and different types of database architectures.

In the rest of the Chapter, we briefly overview different types of multi-core CPU architectures, some of which are used during research exploration in this thesis. We also overview different types of standard parallelization libraries being used during workload parallelization on these CPU architectures. Finally, we provide a brief survey of state-of-the-art research addressing multi-core CPU utilization in different types of database system workloads.

2.1 Relational databases

Databases offer an elegant solution to data management problems. They are used to store and retrieve data efficiently. The database schema imposes a structural format on the stored data. The schema describes different types of data and the relation between them using different forms of constraints. Based on the nature of the schema the databases can be categorized into relational, graph, nosql databases, and so on. Relational database systems are one of the oldest and the most popular type of database systems [51]. In this thesis we focus on the relational database management systems.

A relational database schema consists of different data attributes of different data types stored in a table representation as shown in Table 2.1. When the data is loaded in the database each data record is matched against the schema's attribute

data types, before being stored in the database. The query interface allows to query the stored data for different types of analytics.

Table 2.1: **A relational schema.**

Id (Int)	FirstName (Varchar)	LastName (Varchar)
1	Mrunal	Gawade
2	Pham	Duc
3	Eleni	Petraki

The database systems are a complex piece of software with different components. The main components are the query language front end, the plan generator, the query optimizer, and the query execution engine. We summarize each of these components in brief next.

One of the most popular query languages is SQL (Structured Query Language), which provides a standard interface for querying the data. An example SQL query looks as follows.

Select Column from Table where Column_data=1;

Most database systems use a relational algebra representation of the input SQL queries, where the SQL query is converted into a relational plan representation. The relational algebra plan representation for the query is shown in Figure 2.1.

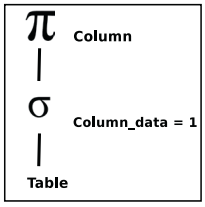

Figure 2.1: **Relational algebra plan**

A relational algebra query plan represents logical ordering of different types of relational algebra operators such as scan, selection, join, group-by, etc. These operators are implemented using different algorithms, suitable for a particular logical plan representation, depending on various characteristics such as the type of data, the distribution of data, size of the data, presence of auxiliary structures such as the indexes, the hardware characteristics, etc. For example, depending on the characteristics of the data being accessed a select operator could use a scan select, or index select operator algorithm.

Hence, depending on various factors such as the physical operator's algorithm being used, join ordering of operators, and other possible operator orderings there are combinatorial choices for the available physical plans.

The query plan optimizer is one of the most critical components of a database system, as it has to discard bad plans and choose a near optimal plan amongst all possible plans. The query plan optimizers are categorized into cost based optimizer and heuristic based optimizer. A cost based optimizer uses a cost model to decide

the execution cost of various possible plans. A heuristic optimizer uses different heuristic rules to select heuristically optimized plans. Cost based optimizers are often complicated and can result in sub-optimal plans due to errors in estimation of the operator cardinalities. Heuristic optimizers in comparison are simple and often do not use operator result estimation cardinalities.

The query execution engine is responsible for the execution of the final plan selected by the query optimizer. Most execution engines are interpreter based. However, a recent trend is to use just in time compiled plans for execution [116]. The engines can be further classified into tuple-at-a-time, operator-at-a-time and vectorized execution. We elaborate about them further in Section 2.3.

The relational database systems can be further classified based on the types of the workloads they support. The types of queries used in a database system determines the workloads under use. Traditionally there are two types of relational database workloads, transactional and analytical.

2.1.1 Transactional workload

The transactional workloads touch only a few records from the stored data per query. A common example is a banking transaction, where the users bank balance gets updated on credit / debit transaction. Since, transactional workload queries touch a single record, and execute in a few milli-seconds mostly, the transactional workloads are bench-marked on the basis of their throughput, that is the number of transactions completed per second. TPC-C [1] is a standard benchmark used for transactional bench-marking, also termed as OLTP (On line transactional processing) benchmark.

2.1.2 Analytical workload

On the contrary in an analytical workload queries touch millions of records to aggregate data from all the available records, to derive certain statistics. Aggregation is mostly required only on a few columns of interest. Hence, analytical queries take a longer time to execute and response time optimizations is one of the main challenges in such workloads.

Benchmarks are used for a relative performance comparison of different database systems. Transaction processing performance council organization (TPC) is an organization that is responsible for different benchmarks used in the database research. TPC has benchmarks for both analytical systems and transactional systems. The data generator tool allows generation of different sized data. TPC-H [2] is the benchmark for analytical systems. It has 22 analytical queries of different complexity. Most analytical database systems use TPC-H queries for performance comparison. TPC-DS is a relatively new benchmark with 99 queries that are more diverse. In this thesis we prominently use TPC-H benchmark. TPC-DS benchmark is also used some times to bring more diversity to the experiments.

[1] www.tpc.org/tpcc
[2] www.tpc.org/tpch

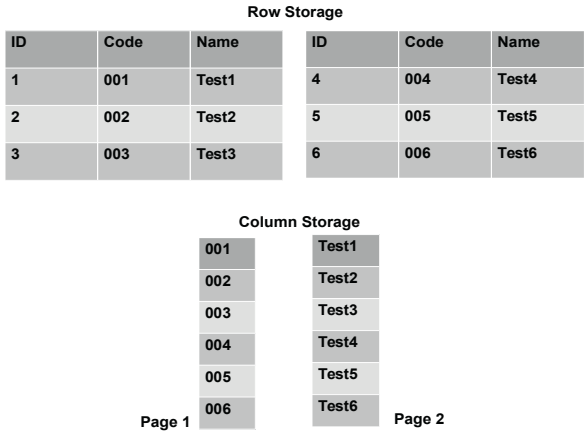

Figure 2.2: **Row store vs Column store.**

Having over-viewed relational database logical model and its architecture components, next we provide a brief overview of the categorization of relational database systems based on their storage layout format.

2.2 Database architectures and storage types

The layout of the stored data in the database systems gives rise different types of storage architectures namely, 1. N-ary storage (row-store) 2. Decomposed storage (column-store) 3. Partially decomposed storage (hybrid-store).

2.2.1 Row-store

In a row-store the entire record (tuple) consisting of multiple attributes (columns) is stored adjacently in a single page [123]. This type of storage architecture is good for transactional systems which touch only a single tuple using indexed look-ups. Bringing a single page containing the relevant record into cache is efficient.

However, if a row-store architecture is used to process analytical queries, then a lot of unnecessary data in the columns not needed for analysis is also brought into the CPU caches for processing. It not only wastes the precious memory bandwidth, but also disturbs the data locality in caches.

2.2.2 Column-store

On the contrary, in a column-store architecture each individual attribute (column) of a tuple is stored in its own storage scheme [52]. Analytical queries use only a few attributes (columns) from the complete schema. When queried, only the

attributes required are brought into CPU caches for further processing. Column-store architectures also help in better compression of the stored data as most data tend to be co-related, for example, a column storing date type, will repeat the year and the month for many records. Memory bandwidth is not wasted, as only the needed data is brought into the CPU caches. Column-stores also allow simple loop based processing while keeping the data in CPU caches, and improving the CPU performance in terms of branch misprediction logic, cache misses, etc. Hence, column-store systems are the predominant vehicles for analytic data processing.

Tuple reconstruction

As processing in a column-store occurs on individual columns, projecting the final result calls for inter column access. The process of formation of the complete record is known as the tuple reconstruction problem. Tuple reconstruction can be done immediately after an operator execution or can be postponed till the final projection operation. Accordingly there are two types of tuple reconstruction methods named as eager and lazy tuple reconstruction [84]. Tuple reconstruction could add considerable cost depending on the method being used.

2.2.3 Hybrid-store

Row-stores are optimal for transactional workloads, while column-stores are optimal for analytical workloads. However, not all workloads have such distinct characteristics. Such mixed workloads (OLTP / OLAP) can show a mixed behavior between row and column store architectures. Such workloads benefit from using a partially decomposed storage model (PDSM) architecture (hybrid-store) [73]. In this model database schemas are decomposed into (multi-attribute) partitions such that a given workload is supported optimally.

Having overviewed the database architectures on the storage layout, next we briefly categorize them based on their relational processing models.

2.3 Database architectures and relational processing model

Database architectures could be also categorized based on their relational processing model. This aspect concerns the execution engine paradigm itself. Most database systems use a pipe-lined execution engine where operators pass the data in a pipe-lined manner in their logical ordering in the plan. Accordingly the execution engines are classified as follows.

2.3.1 Tuple-at-a-time execution

This model was introduced by the Volcano system [70] and is also known as the Volcano style query processing. Most database execution engines use a variant of Volcano style query processing. It involves an open-next-close iterator model, where an operator calls the *open* iterator, and then the *next* iterator to start receiving the data from the next operator in the execution plan pipeline. However, this is a

highly inefficient model from CPU performance as unpredictable function pointer chasing during operator iterations, leads to branch mispredictions, pipeline flushes, etc.

Row stores were popular with disk based system and used open-next-close iterator model to access tuples, since the latency of accessing a single tuple from disk is high, so CPU performance did not matter much. However, the storage architecture shifted to the column stores where CPU performance started to matter as the column stores use in memory query execution engines [41], where the data is stored in the memory. Hence, the disk access latency gets replaced by memory access latency, and the CPUs memory access latency thus becomes important. This gave rise to the bulk processing model, which we describe next[94, 41].

2.3.2 Operator-at-a-time execution

One of the early column stores MonetDB, uses the operator-at-a-time execution model. This type of processing is also termed as bulk processing. Here precompiled primitives are static loops without function calls that materialize all intermediate results. In this model an operator executes completely before another dependent operator in the pipeline starts execution [39]. Complete materialization however increases the intermediate data size and uses a lot of memory, outweighing CPU efficiency. To improve on this deficiency, vectorized bulk processing is used, as described next.

2.3.3 Vectorized execution

To avoid the penalty of complete materialization, a vector-at-a-time execution engine was proposed in the MonetDB-X100 system [42], which was later commercialized as Actian database system. Vectorwise uses an open-next-close based pipelined execution engine with vectorized execution where instead of fetching a tuple at a time, a vector-at-a-time is fetched. Vectorwise uses CPU caches effectively by keeping vector sizes cache conscious and thereby avoiding memory access latency.

Having looked at different database architectures and their query processing paradigms, next we provide a brief overview of different query execution performance improvement opportunities.

2.4 Query execution bottlenecks

Query execution response time is one of the main metrics to measure the query's execution performance. The query execution could get hampered due to different problems arising due to inefficient operator algorithms, lack of indexes, resource constraints at the CPU, the memory and the network level. The query execution time can be improved in two ways, software level improvement, and hardware level improvement.

2.4.1 Software level improvement

In software level improvement the focus is on the design of new algorithms for database operators, indexing techniques, query optimization techniques, etc. that could reduce the query execution time. A lot of research is focused onto the optimization of different join operators, scan operators, group-by operators. We overview some of it in the context of multi-core CPUs in Section 2.8. In memory indexing strategies such as adaptive indexing (cracking) [85] help mitigate the cost of building an index. Imprints [136] allows to index in memory data at cache line granularity. Adaptive query optimization [58] techniques help to correct problems in cardinality estimations to create more optimal plans for execution.

Query execution bottleneck problems are difficult to identify. Hence, different debug tools are used to get in-depth insights into the possible query execution problems. Visualization tools help to identify the plausible problems.

A standard way to improve query execution performance is using plan parallelization for the underlying multi-core CPU architecture.

Query parallelization

Traditionally there are two types of query parallelization techniques. Intra-query parallelization and inter-query parallelization. During intra-query parallelization individual operators in a query plan operate on partitioned data such that operators working on independent data can operate in parallel. On the contrary during inter-query parallelization multiple queries execute on individual cores in parallel, such that each query executes only on a single core. This is a form of parallelization where though individual queries execute serially on each core, as a whole multiple queries execute in parallel on different cores.

2.4.2 Hardware level improvement

Another important field of interest is how the software behaves in co-ordination with the emerging hardware technologies. New hardware such as the multi-core processors, the memory technologies such as NVRAM, the storage technologies based on flash, the network technologies based on new network inter-connects keep on emerging. Hence, software evolves in tandem to make the most of the new hardware features. A good example of this trend is the new processor architectures with features such as super-scalar execution, out-of-order execution, deep pipelining, branch-predictor logic, non-uniform memory access (NUMA) sockets and their interconnects, different level of caches, hardware threading, etc. Database software algorithms need to be able to effectively utilize all the available features in the hardware to be able to effectively drive the query execution.

In this thesis we focus on query execution improvement by query parallelization in the multi-core CPU context.

2.5 Multi-core CPU landscape

CPU architectures continue to evolve from the single core CPUs to multi-core CPUs, and multi-socket multi-core CPUs to many-core CPUs. Having done a brief overview of the relational database systems background, we now provide a brief overview of the multi-core CPU architecture landscape, some of which we use in this thesis.

2.5.1 Multi-core CPUs

Multi-core CPU systems have multiple CPU cores on a single socket. The number of cores per socket varies, however, 4 cores per socket are common in desktop level processors. We use Intel Xeon based multi-core CPUs in this thesis, which have private L1, L2 caches and shared L3 cache.

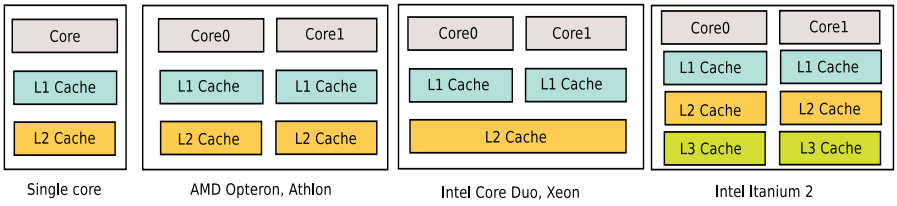

Figure 2.3: **Different multi-core CPU architectures.**

Figure 2.3 shows such sample cores with different CPU architectures. Each core has different hierarchy of caches (L1 / L2 / L3). Cache is very small sized, fast memory with very low access latencies compared to the main memory access latency. The approximate access latencies are L1 = 4 cycles, L2 = 10 cycles, L3 = 60 cycles, the main memory = 120 cycles. In modern processors both L1 and L2 cache are usually on the core itself, whereas the L3 cache is shared across all the cores. Modern processors also have integrated graphics processing unit on the die.

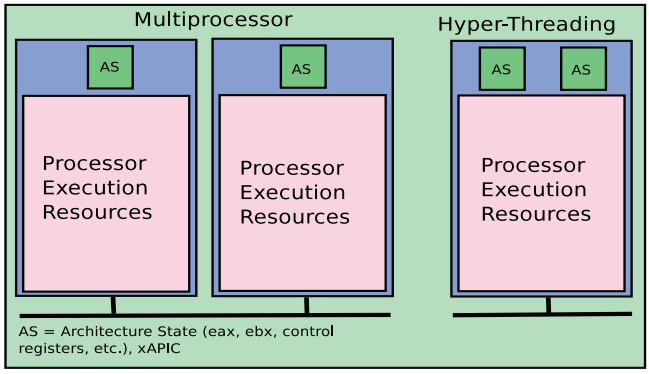

Figure 2.4: **Hyper-threading architecture.**

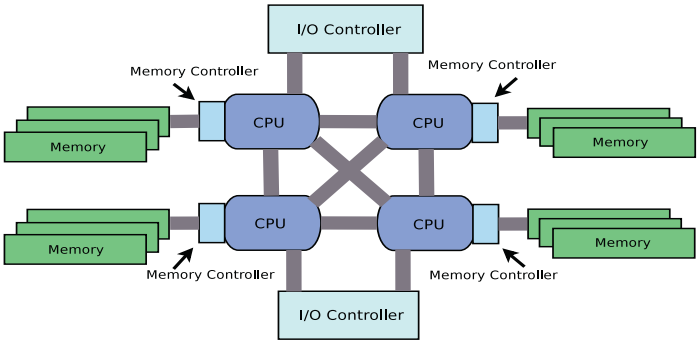

Figure 2.5: **4 socket NUMA architecture.**

The number of cores also appear to increase due to simultaneous multi-threading (SMT). SMT is a technology for emulating multiple logical cores per physical core, so that more than 1 thread can execute per core. Hyper-threading (HT) is a special form of SMT in Intel processors where the register state per core is duplicated so that one more thread can execute per physical core. Figure 2.4 shows a representative diagram of hyper-threaded CPU compared with multiprocessor CPU. Hyper-threading however need not improve performance always and is good when there is overlap of memory bound and computational bound threads per core.

2.5.2 Multi-socket multi-core CPUs

The physical limitations restrict putting more number of cores on a single socket due to problems such as heat dissipation, and transistor size. Hence, to accommodate more number of cores, more number of sockets are added in the system, resulting in a multi-core multi-socket system. Each socket is connected with other sockets through an interconnect technology by the CPU vendor. Quick path interconnect (QPI) is the Intel's proprietary interconnect technology, whereas Hypertransport is the interconnect technology of AMD processors. Each of these interconnect technologies use different topology such as direct connection (single hop) and indirect connection (multi-hop).

Figure 2.5 shows 4 sockets directly connected with all other sockets. There can be configurations where each socket is connected only with its direct neighbors, as we use in Chapter 7. Each socket is further connected with its own memory module. The term NUMA arrives from the non-uniform memory access that arrives due to asymmetric memory access by each socket. For example, consider in the example configuration in Figure 2.5, the local memory access of a socket has much less latency as compared to the remote memory access of a neighboring socket, as the remote memory access has to go over the interconnect.

Intel announced NUMA compatibilities for its x86 and Itanium servers in late 2007, while AMD announced it with Opteron processors in 2003. 4 socket CPUs

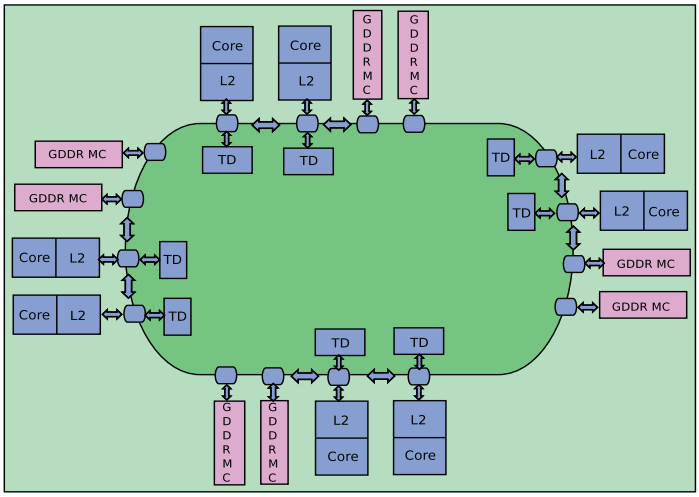

Figure 2.6: **Intel Xeon Phi architecture.**

are the most common at present in the server systems. NUMA can be also viewed as a form of cluster computing. Virtual memory support in cluster based systems could allow implementation of NUMA in software. As NUMA affects memory access performance, softwares need to provide certain optimizations from scheduling of threads close to the in-memory data. Linux provides support for NUMA based memory allocation at the kernel level. Microsoft Windows and Windows server 2008 provides NUMA support. OpenSolaris models NUMA support with lgroups. Java 7 has support for NUMA aware memory allocation and garbage collection.

2.5.3 Many-core CPUs

Many core systems are the next evolution in the area of multi-core systems. These are the systems with many power efficient cores, which are connected through a common interconnect. Intel Xeon Phi family of co-processors represent the many core architecture (See Figure 2.6). Each individual core is based on the old Pentium architecture with a frequency of around 1GHZ. The number of cores are around 60. Each core can support 4 physical threads, hence a total of 240 threads can be hosted on a 60 core architecture. Each individual core is simpler in architecture compared to the complexity of Xeon cores. Each core has 2 pipelines, hence for optimal usage of the logical units, a minimum of 2 threads are required per core. Xeon phi architecture is code named Knights Corner architecture. The cores support in-order processing compared to out-of-order processing of the Xeon cores.

Xeon phi architecture is designed keeping in mind the HPC workload with GPU architecture as the main competitor. Each core has a 512 bit SIMD vector unit, 32 KB L1 cache, 512 KB L2 cache. The L2 cache is kept coherent through combination with the interconnect. The memory bandwidth ranges between 250 MB/s to 320 MB/s. Xeon phi in the current architecture is used as a co-processor by con-

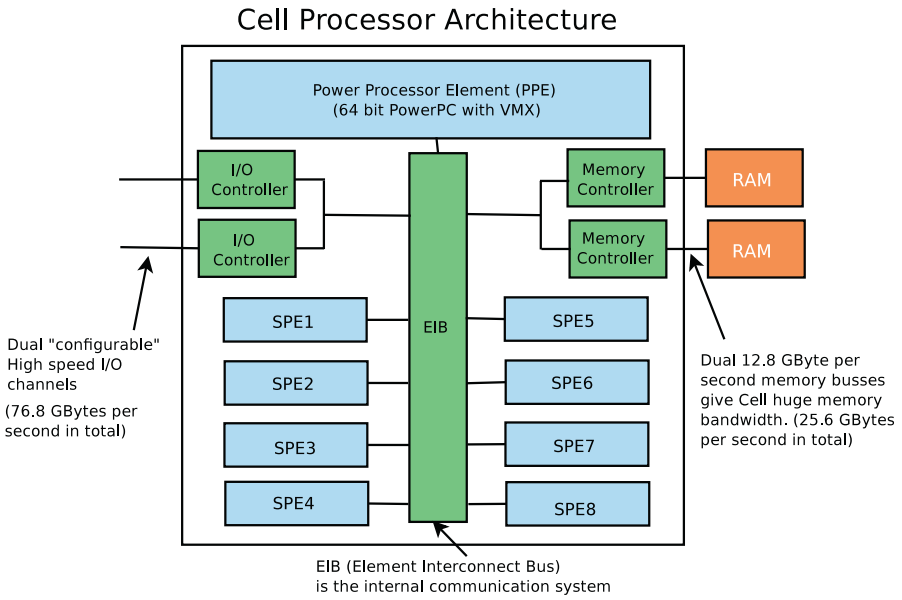

Figure 2.7: **Cell processor architecture.**

necting it over the PCIe bus. Hence, the PCIe bandwidth limitation comes into picture while doing the data transfer between the host and the co-processor. Database oriented workloads tend to be large in size and hence could get constrained by the amount of data being transferred over the PCIe bus.

Intel supports different modes for computation such as native mode, offload mode, silk shared mode. Depending on the complexity of the job, and the amount of data to be handled we can choose one of the modes. In the native mode entire data fits in the device memory and entire computation occurs on Xeon-Phi. In the offload mode, part of the computation gets offloaded along with data to Xeon-Phi. While in Silk shared mode the offloading uses virtual shared memory space between host and Xeon-Phi.

2.5.4 Heterogeneous core CPUs (Cell processor)

Another interesting architecture to consider is when heterogeneous cores are involved. Cell processor is a good example of such an architecture (See Figure 2.7). It contains 1 Power-PC Processing Element (PPE) core, and 8 Synergistic Processing Elements (SPE) core. All SPEs are identical to each other but SPE and PPE are different.

PPE is a simple core whose functionality is like a general purpose microprocessor. It has a 64KB L1 cache and a 512KB L2 cache and features SMT, similar to Intel's Hyper Threading. It also has an in-order core, which was present in original Intel Pentium architecture. PPE is a 2-issue core, meaning at best it can issue 2 instructions simultaneously. PPE is designed to run at very high clock speeds. Since

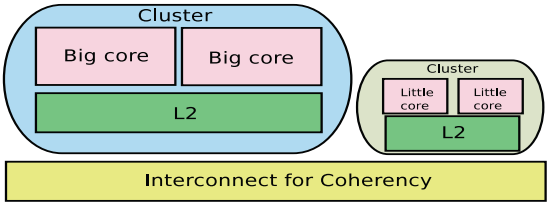

Figure 2.8: **big.LITTLE ARM based mobile processor architecture.**

Cell processor was designed from gaming market as a focus, the performance of individual cores is not as high as a generic purpose micro-processor.

Majority of the die area is covered by SPE which are like PPE in terms of their generic processing nature, but with a more focus. SPE is more simple than PPE, and is not as generic as PPE. It does not have a cache but has 256KB of local memory. Each SPE also has a total of 7 execution units, including one integer unit, so the SPEs can perform integer math as well as SIMD floating point arithmetic. It also can issue 2 instructions simultaneously. It has no branch predictor, and relies solely on the software branch prediction.

Some of the applications of the Cell processor are in the video processing card, blade server, PCI express board, console video games, home cinema, super-computing, cluster computing, and distributed computing.

2.5.5 Heterogeneous core CPUs (Mobile based processors)

Driven by the consumer demand, the number of mobile based devices keep on increasing per year. Many of these devices use Android based operating system, and use ARM based processors, which use heterogeneous multi-cores with big.LITTLE architecture.

The big.LITTLE architecture consists of two sets of cores. The big cores are powerful cores, and the LITTLE cores are less powerful cores. The cores differ in their power consumption and clock frequency. ARM's marketing promises up-to 75% savings in power usage for some activities.

Typically only a single set of cores are active at once, but since all cores have access to the same memory area the workload can be swapped back between big and LITTLE dynamically. There are three ways in which processor cores can be arranged in the big.LITTLE design by the scheduler.

Run state migration

- **Clustered switching -** In this mode the processor cores are arranged in identical sized clusters of big / LITTLE cores. The system changes the scheduled cores based on the load activity. The scheduler uses only a single cluster at a time. All relevant data is passed through common L2 cache, the first cluster is powered OFF and the other cluster is activated. Cache coherent interconnect is used. This is the most simple design. Samsung Exynos 5 Octa (5410) uses this design.

- **In-kernel switcher (CPU migration) -** It involves pairing of a big and a LITTLE core, which appear together as a single virtual core where only a single core is active at a time. A more complex arrangement can involve more than one big or LITTLE cores depending on the workload. NVidia Tegra 3 SOC uses a design based on in-kernel switcher.

- **Heterogeneous multi-processing -** The implementation with most power consists of usage of all cores where threads with high priority / computational intensity can be allocated to big cores, whereas the threads with less priority / less computational intensity such as background tasks are assigned to the LITTLE cores. Samsung Exynos 5 Octa, 7 Octa, and 5 Hexa uses this implementation.

Having overviewed different types of multi-core CPU architectures, next we provide a brief overview of different hardware micro-architecture level parallelization features of the modern processors.

2.6 Types of parallelism

Query parallelism is mostly expressed using query plan level parallelism. However, the parallelism is also supported at the hardware architecture level through different types of computer architecture features, such as instruction level parallelism through instruction pipe-lining, data level parallelism through SIMD, task level parallelism, and thread level parallelism. We describe each of these next.

2.6.1 Instruction level parallelism

Parallelism is supported at the hardware level by using different CPU architecture level features. One of the most prominent feature is instruction level parallelism (ILP), which is a measure of how many operations can be performed simultaneously in a program. It can be further categorized into software and hardware level parallelism. How much parallelism exists in a program depends on the application under consideration. For example graphics and scientific computing programs have large scope for parallelism, where applications such as cryptography exhibit much less parallelism.

Micro-architectural techniques used to exploit ILP are as follows.

1. Instruction pipe-lining - Execution of multiple instructions is partially overlapped.
2. Super-scalar execution - Multiple execution units are used to execute multiple instructions in parallel.
3. Out-of-order-execution - Instructions execute in any order that does not violate data dependencies.
4. Speculative execution - Execution of complete instruction or parts of instructions before being certain this execution should take place.

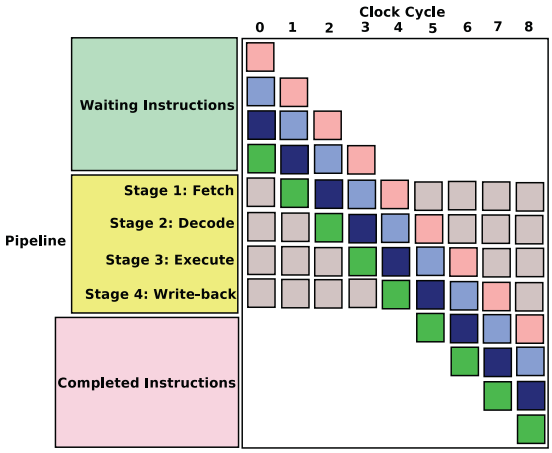

Figure 2.9: **CPU instruction pipeline.**

5. Branch prediction - Used to avoid stalling of instruction execution for due to control dependencies. Used with speculative execution.

Instruction pipe-lining: We briefly describe the instruction pipe-lining technique first. CPU execution unit can accommodate multiple instructions in a pipe-lined architecture. A typical CPU instruction life time consists of the following stages.

1. Instruction fetch
2. Instruction decode
3. Instruction execute
4. Result write-back

While an instruction moves through these different stages one cycle at a time, more instructions can be added to the previous stages, so that the pipeline stays busy with instructions always. Thus, depending on the complexity of the pipe-lined stages, a CPU can support more than 4 instructions in parallel. For the example pipeline in Figure 2.8 the pipeline consists of 4 stages. Thus, per cycle there are 4 instructions active in the execution unit of the CPU.

2.6.2 Data level parallelism

Data level parallelism implies allowing multiple data items to be operated upon by a single instruction. It is also termed as Single Instruction Multiple Data (SIMD) parallelism. For example, consider an example where two 32 byte integers are added using an add instruction. Lets assume that the integers are stored in 32 byte registers. Instead of 32 byte registers if we have 512 byte registers, then 16 integers of 32 bytes each can be stored. An add instruction that is aware of these 512 byte registers hence will be able to add these 16 integers in parallel.

Data parallelism is supported at the processor level using SIMD instructions. Intel supports MMX and iwMMXt, SSE, SSE2, SSE3 SSSE3 and SSE4.x instruction set that is SIMD enabled. Most modern CPUs support SIMD instructions for

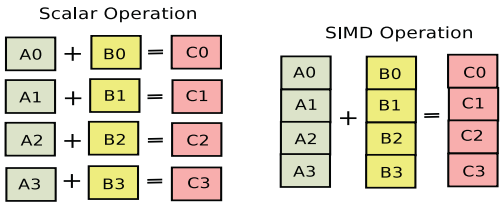

Figure 2.10: **SIMD data parallelism.**

data parallelism. Intel's AVX SIMD instructions process 256 bits of data. Intel's Larrabee prototype microarchitecture includes more than two 512-bit SIMD registers on each of its cores. The 512-bit SIMD capability is being continued in Intel's future Many Integrated Core Architecture (Intel MIC).

An application that may take advantage of SIMD instructions is where the same value is being added or subtracted from multiple data points.

Programming with SIMD might involve following challenges.

1. SIMD might have restrictions on data alignment.

2. Gathering / scattering data from / to multiple locations can be inefficient.

3. Instruction sets are architecture specific, so some processors might lack them.

2.6.3 Task level parallelism

Task level parallelism involves dividing a task into multiple smaller granularity tasks such that the smaller granularity tasks can be scheduled to execute in parallel by multiple processes or threads. Once a particular thread of execution is finished with a particular task, it can start executing another task until all tasks are completed. Task level parallelism involves work-stealing based approaches towards scheduling tasks on different processes or threads of execution. Task is the schedulable entity with respect to the available set of fixed processes or threads of execution.

2.6.4 Thread level parallelism

Thread level parallelism involves multiple threads of execution that work on the divided tasks. For example, consider a column of data being queried during database execution. If there are n threads of execution operating in parallel, lets assume that the column is divided into n equal pieces. Then each thread operates on its individual pieces of data until it finishes its execution. Thus in thread level parallelism the emphasis is on dividing the tasks equally on all the available threads, such that each thread gets its piece of work.

Thread level parallelization of software is further realized through different parallelization system libraries. We provide a brief overview next.

2.7 Parallelism support with system libraries

Thread level parallelism is supported at the system level through different types of subsystems. Some examples being the posix threads (p-threads), OpenMP, threading building block (TBB), and Cilk / Cilk Plus.

These subsystems differ at the level of abstraction they provide for different types of threading level interfaces for programs. For example, pthread library provides detailed api for thread management, whereas the OpenMP library provides pragma based high level directives to manage threads, and takes care of api level details of thread management internally. We describe some of these subsystems here in brief.

2.7.1 Posix Threads

Posix threads is a posix standard for threads, and exists as an execution model independent of language. It provides APIs for creating and manipulating threads. The API implementations is available on many POSIX compliant operating systems such as FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Solaris, and also Microsoft Windows.

Pthread prescribes a set of C programming language types, functions, and constants, defined in pthread.h and a thread library. There are around 100 Pthreads procedure all starting with "pthread_". Some sample APIs are pthread_create, pthread_join, pthread_destroy. These can be categorized into following types.

1. Thread management
2. Mutexes
3. Condition variables
4. Synchronization between variables using read / write locks and barriers.

2.7.2 OpenMP

OpenMP describes an application programming interface (API) for multi-platform shared memory multi-processing programming with C, C++, and Fortran. It supports most platforms, processor architectures, operating systems including Linux, Solaris, AIX, HP-UX, OS X, and Windows.

OpenMP implements multi-threading where a master thread forks slave threads and the system divides a task amongst them. Various preprocessor directives are used to direct the multi-threading support. For example. the following directive forks multiple threads which execute the printf statement.

#pragma omp parallel
$printf("Hello, world.");$

After the execution of parallelized code the threads join back into the master thread, which continues the execution further.

Each thread executes independently, however, work sharing constructs can be used to divide the task amongst the threads so that each thread executes its allocated part of code. Both task and data parallelism can be achieved in this way.

2.7.3 Threading building blocks

Threading building block is a C++ template library by Intel for multi-core processor based parallel programming. It consists of data structures and algorithms that abstract away complications of using threads using native threading libraries such as Posix threads. The library uses the concept of task based allocation to different cores dynamically by the run-time engine. A TBB program creates, synchronizes, and destroys the tasks from a graph based allocation based on algorithms. Tasks are executed with graph dependencies.

TBB uses work stealing approaches for balancing the parallel workload to cores. This allows it to increase multi-core utilization and scale better. Initially the work is divided equally amongst the available cores, if one of the cores finishes earlier, then some of the work from one of the busy cores is dynamically reassigned to the idle core.

2.7.4 Message Passing Interface (MPI)

Message passing interface is a portable and standardized message passing system designed to work in a shared memory and distributed clustered environment. The standard defines different routines that can be used from multiple languages such as C, C++, FORTRAN, Java. There are multiple implementation of MPI available, some prominent ones being OpenMPI, MPICH, MVAPICH, and Intel MPI. It is widely used in high performance computing environment.

MPI is a language independent communication protocol system designed to be used in a clustered environment. It supports both point to point and collective communication. MPI programs are regularly run over shared memory systems, though its main usage is in clustered computing environment.

MPI programs work with processes (also called MPI rank), and it is common to have a process allocated for a single core. MPI uses library routines such as MPI_Send and MPI_Receive to send and receive data in buffered, synchronous, asynchronous mode. Standard data types are supported during data transfer, but MPI also allows custom data type creation.

MPI programs could show different performance behavior depending on the type of setting being used. Different parameters that can be tuned for performance optimizations are type of protocols used (eager / rendezeneous) for data transfer, threshold parameter to chose these protocols, buffer sizes, collective algorithms, etc.

MPI supports PMPI interface as the inbuilt interface to expose the performance profiling options. However, PMPI interface has limitations in terms of the amount

of information it could expose for profiling. A new interface called MPI_T exposes more information in terms of control and performance variables. The control variables expose the possible tuning parameters whereas the performance variables exposes the internals of MPI implementation libraries such as message sizes, buffer sizes etc.

Having overviewed the background on relational database systems, multi-core CPU architecture landscape, and software parallelization system libraries, next we provide an overview of the state-of-the-art research in database systems to utilize multi-core CPU architectures.

2.8 Database parallelization and multi-core research

Database systems are a complex piece of system software with components such as query parser, query optimizer, operator's physical implementation, buffer manager, execution engine, log manager, etc. Different multi-core CPU architectures affect each of these components in different ways thereby affecting the overall query execution performance. In the remaining Chapter we explore state-of-the-art research exploring the related problems.

2.8.1 Multi-core parallelization of operators

As database systems use query interface to extract analytics out of stored data, query plan parallelization plays important role during query execution, to minimize the query response time. Depending on the architecture under use database systems use different techniques such as intra-operator and inter-operator parallelization.

Intra-operator parallelization involves multiple instances of the same type of operator working on different partitions of the data in parallel. Inter-operator parallelization involves different types of operators working in parallel. Intra-operator parallelization is realized in the query plan using the exchange operator mechanism. Most database systems use the exchange operator based parallelization. We provide a brief overview of it next.

Exchange operator

The exchange operator based parallelization was introduced by the Volcano system [68]. Volcano system also introduced the pipe-lined model of query execution where logically ordered operators in a relational algebra plan use open-next-close iterator model to fetch tuples from the dependent operators in the pipeline. During query plan parallelization the query optimizer first selects a near optimal serial plan. This plan is then parallelized by introducing exchange operators at strategic places such that depending on the data partitioning technique used, the operator pipelines are replicated to work on partitioned data, and the resulting tuples of individual operator pipelines are combined using exchange union operators. Exchange union operators thus heavily rely on data partitioning strategies. They allow an easy way for plan parallelization where the logical plan representation does not change much from the original input serial plan.

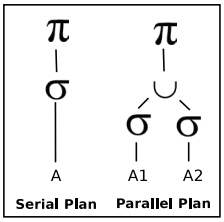

Figure 2.11: **Exchange union operator based parallelization.**

An exchange union operator acts as the pipeline breaker, as it has to gather tuples from different forked pipelines, and pass the aggregated results to the next logical operator in the pipeline. In Figure 2.10 the projection operator starts execution by the open call, followed by the next call which invokes the open call for the selection operator, followed by the next call to fetch tuples from A. When an exchange union operator (U) is introduced to parallelize the selection operation it breaks the above pipeline. Deciding the number of forked pipelines to be combined by the exchange union operator hints at the degree of parallelism of the plan. As the number of CPU cores continue to rise, finding an optimal degree of parallelism thus becomes a crucial aspect of the plan parallelization process, as a sub-optimal plan could result in over / under utilization of CPU cores as a resource, punishing other possible concurrent operations during query execution.

A lot of research focuses on making individual relational algebra operators such as the join operator multi-core efficient. However, in real systems a single operator never executes in isolation, and gets affected by the run-time resource constraints due to other operators under execution. Insights about a single operator execution from multi-core perspective still assumes importance. We explore the relevant research for fundamental operators such as scan, join, and group-by.

Join operator

Join operator is one of the most important operators and hence is also the most researched operator from multi-core parallelization perspective. It comes in different flavors based on the type of join algorithm used during implementation. Some common implementations are hash join [59], sort-merge join [69], radix join [109], etc. A new research direction proposes that the modern hardware is smart enough to make the join operator hardware oblivious [36]. In contrast there is research that focuses on a hardware conscious join operator [32]. How to do join operation efficiently on massive data sets is also another interesting research direction. We briefly overview some of these research directions next.

One line of argument is to make the join algorithms hardware oblivious [36]. Especially tuning the partition phase, where the data is made to fit in the caches, is not needed as the modern hardware can efficiently handle the memory hierarchy latency differences. The algorithm builds a shared hash table on the build relation using millions of hash buckets keeping latch synchronization overhead amongst

the worker threads minimal. The probe phase accesses the hash table in read only mode, hence does not require synchronization. This algorithm is robust to sub-optimal parameter choices by the optimizer and does not need input characterization knowledge. It uses intrinsic hardware optimizations to handle the skew. The authors use two extreme architectures, the first one being from the Intel Nehalem family and the second one UltraSPARK T2 with 8 simple cores that all share a single cache. Each core can use 8 threads, for a total of 64 threads per processor.

In contrast there is research on tuning of the join algorithms based on the underlying hardware architecture. In [32] authors demonstrate through experimental analysis of various algorithms and architectures that hardware characteristics matter considerably. The paper shows based on the selection effects (relative table sizes, tuple sizes, the architecture in use, use of sorted data, use of SIMD, page sizes, TLB sizes, tuning of the implementation, etc.) the hardware characteristics play a significant role. According to the authors hardware obliviousness focuses only on a small set of parameters. The authors verify the claims by [36] and conclude that the shared hash table is built on a pre-sorted data making it easier to avoid cache and TLB misses. The authors also can not verify the claims of the role of SMT in hiding the cache misses. Finally the authors claim the fastest implementation of the radix join with close to 200 million tuples per second. They use 4 different hardware architectures comprising of older Intel Nehelam to the newer Intel Sandy Bridge, AMD Bulldozer, and Sun UltraSPARK T2.

Continuing the hardware consciousness line of thought, the authors in [95] explore the effectiveness of sort vs hash based join algorithm and conclude that SIMD is still not good enough to tip the decision in the favor of sort-merge join instead of commonly used hash join algorithm. Their analytical model concludes that in future as SIMD becomes wider (512 bit), sort-merge join will perform better. For hash-join to perform better in wider SIMD environment, it needs hardware based support for efficient scatter and atomic vector operations. Authors also implement the fastest hash join and sort-merge join algorithm on latest hardware platform and report 17x time improved performance than best reported hash-join performance for CPUs and 8x faster for best reported GPU performance. They show at present hash-join is superior on their platform compared to sort-merge join.

In [28] authors design NUMA aware massively parallel sort-merge (MPSM) join algorithm that scales linearly with the number of cores, and outperform hash-join implementations. It uses partial partition-based sorting. Compared to the classical sort-merge join, MPSM algorithm does not rely on a hard to parallelize final merge phase to create a complete sort order. Authors show how the Wisconsin hash join [37] which uses a shared hash table during build phase by using latches gets affected due to NUMA accesses, and so is the classical radix hash join [108], compared to the MPSM NUMA aware sort-merge join.

A lot of work focuses on NUMA related aspects of operator designs. In citelang2015massively authors focus on designing NUMA aware massively scalable hash-join. They do this by optimizing the parallel hash table construction via a lock-free synchronization mechanism based on optimistic validation, while also considering an optimized NUMA aware storage layout for the hash table. In [106] authors emphasize the need of being NUMA aware during fundamental op-

erations such as data shuffling. In [48] authors discuss efficient implementation of merge-sort algorithm from SIMD (128 bit) perspective. In [49] authors describe an adaptive aggregation algorithm that decides the best aggregation strategy to use in chip multiprocessor environment based on sampling.

Multi-core CPUs and multi-socket CPUs will continue to evolve in coming time with faster interconnects, different memory hierarchies, different cpu core architectures, integrated CPU-GPU co-design architectures. How to utilize the existing database system architectures and individual components of these architectures to utilize the new multi-core CPU architectures will always provide for new research opportunities.

2.8.2 Cell processor parallelization of operators

In [81] the authors test the feasibility of vectorized database engine on a cell processor. Some prominent observations are 1. All computations should be performed using SIMD on Synergistic Processing Elements **(SPE)**. 2. All performance critical if-then-else branches need to be eliminated as SPEs don't have a branch predictor, and there is high branch penalty. 3. The limit on code size is hard as SPEs have a limited program code size which should not exceed 256KB. Authors experiment with tuple-at-a-time Volcano style processing, MonetDB column-at-a-time processing, and MonetDB-X100 vectorized processing. In all three models of execution the interpreter is executed on Power Processor Element **(PPE)**, whereas the data processing is executed on the SPE. A performance comparison of TPC-H Q1 shows vectorized execution on cell processor performing up-to 20 times faster than the Itanium2 processor. Though cell processor exhibits an interesting architecture they are discontinued. However, the lessons learned from experience using it, should provide new research directions on related architectures of future.

2.8.3 Many-core parallelization of operators

Intel's many-core platforms are primarily used in high performance computing workloads. Use of it in database workload is being explored as an active research area. In the existing work [89], the authors compare the hash join performance on Xeon-phi against the multi-core CPUs from hardware oblivious and hardware conscious characteristics perspective. A primary finding is that architectural features and software optimizations have different behavior on Xeon-phi compared to multi-core CPUs, which asks for new optimizations and tuning on Xeon-Phi. For example, a much larger performance improvement is observed on Xeon-phi by tuning prefetching, TLB, partitioning, etc, than those on multi-core CPUs. Another observation is that hardware oblivious algorithms can outperform hardware consciousness algorithms on a wide range of parameters.

One of the prominent features of Xeon-phi is the 512 bit SIMD registers. The presence of these registers allows hardware assisted data parallelization, where the same instruction operates on multiple data items. Authors in [121] provide a detailed algorithmic setup for vectorized designs and implementation of different common database operators such as join, selection, sorting using advanced SIMD

operations such as scatter and gather. The new vectorized designs are shown to be at least an order of magnitude faster than the state-of-the-art scalar and vector designs. They also show energy efficiency benefits of such designs.

The attempts to port a complete database execution engine on Xeon-Phi are not successful as Xeon-Phi has a limited device memory and the data needs to be transferred over the PCIe bus for any accelerated computation. As Xeon-Phi is a relatively young architecture, there are still many open research problems. Towards the end of the thesis we show some preliminary work in the related context.

2.8.4 Query optimizer parallelization

Traditionally a cost based or heuristic based optimizer is used to generate parallel plans. However, the cost based optimization time using dynamic programming technique for finding join order of more than 10 tables can increase very rapidly. Authors in [75] use a novel technique to find the join ordering of more than 10 tables by parallelizing the query optimization process on multi-core processors. They also introduce a novel data structure called skip vector array that significantly reduces the generation of in-feasible join partitions. This parallel join enumeration algorithm with skip vector array outperforms the traditional generate-and-filter dynamic programming based algorithm by up-to two orders of magnitude for star queries. Query optimization is a challenging research area in itself. When combined with multi-core parallelization the research challenge complexity of query optimization parallelization grows further.

2.8.5 Hardware oblivious parallelization systems

Ocelot [80] takes an hardware oblivious approach to generate hardware specific parallel plans. It uses a predefined set of hardware oblivious operators that are compiled down to actual hardware at run-time. It uses OpenCL based parallel programming environment which compiles plans for CPU and GPU specific operations. Using hardware specific tuning for individual hardware, generates efficient database code, however, this approach is not portable and scalable with the increasing number of different hardware types. Ocelot is implemented in MonetDB and the comparison on TPC-H query execution with native MonetDB execution, shows Ocelot performance matching on CPU, and exceeding on GPU. As diverse hardware emerges, having an optimal database system implementation matching each type of hardware could be really difficult from a portability perspective. Hardware obliviousness as exhibited by Ocelot opens up interesting new research directions in aligned fields such as cost model optimizations pertaining to individual hardware types.

2.8.6 Multi-core scalability of new analytical systems

New commercial systems such as IBM DB2 BLU [124] and HyPer [93] are examples of grounds up approaches of new database architectures with a focus on multi-core scalability for analytical workloads. Both systems use a kind of work-stealing

based scheduling approach for dividing work amongsts worker threads, while providing flexibility towards degree of parallelism of plans. Traditional approaches use the exchange operator based parallelism baked in the query plan at compile time itself. It makes controlling the work division at run-time difficult as no guarantees for multi-core scalability could be provided based on the run-time resource variations due to concurrent workload. The dynamic scheduling of work amongst worker threads in Hyper allows provision for controlling the degree of parallelism. IBM DB2 BLU [124] uses specially designed data structures that are multi-core scalable, while, Hyper uses just-in-time compiled plans for efficient execution. As the individual components of these new systems are designed from grounds-up for multi-core scalability, they give rise to new interesting research problems in areas such as the operator algorithms, the query optimizers, the execution engine efficiency in terms of vectorization vs compilation, etc.

2.8.7 Multi-core scalability of transactional systems

Relational database workloads can be categorized into transactional and analytical workloads. While the focus of this thesis is on analytical workloads, there is considerable research exploring the effect of multi-core architectures on transactional workloads. The system components, such as log manager, transaction manager, buffer manager, lock manager etc. get considerably affected by the multi-core architecture scalability problem. We review some relevant research next.

Shore transaction system scalability

Shore is an open-source storage manager designed in the era when multi-core CPUs did not exist. Authors in [91] describe how Shore and other open-source storage managers such a BerkeleyDB, MySQL, and PostgreSQL face severe scalability issues with the latest multi-core architectures. Shore-MT is a new multi-core scalable storage manager, developed by removing critical bottlenecks from individual components of Shore. The new system is called Shore-MT.

Shore exhibited severe multi-core scalability problems in individual system components such as the buffer-pool, lock, log, and the transaction manager. One of the main areas of contention in most of the component managers was a single global mutex protecting the entire hash table. It was replaced by a mutex per hash bucket. The main principles applied were shortening or removal of critical sections, elimination of hotspots, and using the right synchronization primitives.

Hardware islands and transactional systems

In [122] authors treat multi-socket systems as hardware islands, in which a single or multiple database systems could be made to run on a single socket. They test different deployments of database systems on different multi-socket systems from shared-everything to shared-nothing deployments. Traditional shared-everything systems under-perform on modern multi-socket hardware as they incur too much overhead due to excessive communication between various threads, and contention

amongst threads. On the contrary shared-nothing systems face challenges in execution due to higher costs when distributed transactions are involved (if the communication occurs between slower links of CPU sockets), and load imbalance due to skew. Authors explore the impact of perfectly partitionable and non-partitionable workloads, with and without data skew on shared nothing deployments of various sizes and shared everything deployments. The system being used is Shore-MT.

2.8.8 Open research problems

This research overview shows that a large fraction of multi-core architecture research targets specific database operators, such as the join operator from scalability perspective. Although understanding the operator's behavior in isolated execution case is important, the behavior changes during query execution as other operators execute concurrently. As new multi-core architecture arrive in the market, testing the feasibility of existing algorithms on these new architectures remains a focus point for database systems. A lot of research also targets transactional systems's scalability in the multi-core context. When new systems are built up from scratch for using the new hardware architectures optimally, they open up new research opportunities, as these systems use novel data structures, algorithms, architectures to work optimally with the underlying hardware architecture.

However, there is a scope for interesting research exploring the effects of multi-core architectures on the query execution performance of existing analytical systems, in a holistic manner as well. This is partially a result of the complex inter-dependence of different database system components that play an effective role during the query execution. Some prominent components are the query optimizer, the interpreter, individual operator's degree of parallelism, run-time resource availability in terms of CPU cores and memory, etc.

Identifying the degree of parallelism of the query plan using feedback based mechanism involves interaction with all the above components. In the rest of the thesis we explore research questions that are aligned in related fields, and that take into account the role of different multi-core architectures from the analyical query execution perspective in the existing database system architectures.

2.9 Conclusion

Multi-core CPUs come in different configurations and form a vast landscape. Understanding their hardware characteristics to design optimal performing software is a challenging task. In this Chapter we provided a brief overview of many such different types of multi-core CPU architectures, along with different software subsystems that can be used to leverage the parallelism offered by the multi-core hardware. We also provide a brief overview of multi-core aligned research in the database system context, and list some open problems. Making optimal use of hardware with abundant number of cores needs efforts on multiple levels during software execution stack. In the remaining of this thesis we explore several possibilities for such hardware-software co-design.

Chapter 3

Query parallelization analysis through graph visualization

"If you can not measure it, you can not improve it." - Lord Kelvin

Effective tools are crucial to identify execution performance issues in the database systems. Most existing tools use textual representation of the performance data, thereby limiting their usability. A visualization approach however can provide much better insights. Execution plans often use a graph representation. The presence of multi-core CPUs make these graphs grow even bigger due to plan parallelization. Plan representations vary based on the system under use. The performance troubleshooting tools hence are tied up with the system under consideration.

This Chapter [1] presents our effort to improve the analysis toolkit for database query execution plans. The Stethoscope is a visual tool to inspect and analyze query execution performance, both online and offline. It provides a convenient visual interface, capitalizing the data-flow graph representation of a query execution plan augmented with query execution trace information. Stethoscope improves the error prone and time consuming activity of analysis of textual execution trace. It helps in understanding where time goes, how optimizers perform, and how parallel processing on multi-core systems is exploited.

3.1 Motivation

Understanding database query execution traces is one of the most complicated issues in a database system. Their analysis is needed to understand and to achieve optimal query execution performance. Queries vary in their complexity and so do their plans. While most systems use a straightforward physical algebra representation for plans, columnar systems like MonetDB use an alternate representation, making plans very large, as they directly encode parallel processing steps.

[1] This Chapter is based on the publication "Stethoscope: A Platform for Interactive Visual Analysis of Query Execution Plans", In Proceedings of VLDB 2012.

Nevertheless, a query execution trace is a good starting point to reflect upon the run-time behavior. In offline mode, the steps taken can be inspected in detail for unanticipated behavior. In online mode, it can provide insight in the total system behavior, e.g. influence of concurrent processes competing with the resources. Performance analysis tools are often specific to the DBMS, because hooks deep in the system are required.

We present Stethoscope, a platform to analyze [10] query plans and their execution traces. Each query plan shows a data-flow dependency, which allows it to be represented as a directed graph. Such graphs could use a dot file format representation [6]. In this graph representation, a node corresponds to an operator and edges between nodes represents the data-flow dependency between them.

The query plan is executed by the database interpreter and in turn get reflected in the form of an execution trace available for online/offline inspection. Stethoscope combines both into a powerful tool, which animates the execution trace and provides navigational access to the portions of interest in the plan. This way, it can be used to monitor long running queries and performance bottlenecks in the kernel.

3.2 Contributions

Searching for performance bottleneck patterns in an execution trace using text-based tools is an error prone and time consuming activity [74]. Visualization tools hence are crucial during parallelized query execution and form an important component of the research ecosystem for parallelized query execution. Stethoscope improves the performance bottleneck resolution abilities significantly by using visual representation of query execution trace. It is the first tool of its kind to provide both online and offline analysis ability and has inspired similar tools in systems like HP-Vertica [137] and SAP [132]. It provides an easy way to understand the performance execution problems in query execution plans, using the following features.

- Interactive animated navigation in complex query plans. Being able to zoom in and zoom out and navigate on specific portion of the graph.
- Color coded monitoring of query execution state changes. Color coding allows easy identification of the bottleneck operators during query execution.
- Run time analysis of execution states using debug window, tool tip text, etc. These features provide extra functionality from selective fine grained analysis perspective.
- Flexible options for filtering of execution traces.

In the remainder of the Chapter, we provide a brief overview of the targeted database system plans, the architecture of the stethoscope, and the possible use cases for its application.

```
function user.s1_2():void;
  X_3 := sql.mvc();
  X_10 := sql.bind(X_3,"sys","lineitem","l_tax",0);
  X_13 := calc.oid(0@0);
  X_18 := 1;
  X_19 := sql.bind(X_3,"sys","lineitem","l_partkey",0);
  X_20 := algebra.thetauselect(X_19,X_18,">");
  X_22 := algebra.markT(X_20,X_13);
  X_23 := bat.reverse(X_22);
  X_24 := algebra.leftjoin(X_23,X_10);
  X_25 := sql.resultSet(1,1,X_24);
  sql.rsColumn(X_25,"sys.lineitem","l_tax","decimal",15,2,X_24);
  X_32 := io.stdout();
  sql.exportResult(X_32,X_25);
end s1_2;
```

Figure 3.1: **A query plan in operator-at-a-time execution model.**

3.2.1 Outline

In Section 3.3 we describe the architecture of Stethoscope.The work-flow of how to use Stethoscope for doing query execution plan analysis is described in Section 3.4. Different performance troubleshooting use case scenarios are illustrated in Section 3.5. We describe the applicability of the tool in other systems in Section 3.6. The summary is provided in Section 3.7, while the chapter is concluded in Section 3.8.

3.3 Architecture

In this section we elaborate on the database system in use for the query execution performance monitoring and the architectural components of the Stethoscope.

3.3.1 Query execution plans

Stethoscope is being used in MonetDB (default branch change-set c56e636745dd), the open-source columnar database system with operator-at-a-time execution [10], which is predominantly used for OLAP based workloads. Its operators materialize the intermediate data completely, and are represented using the MonetDB Assembly Language (MAL), an abstract intermediate language. For example, after a SQL query is parsed, it is converted into a relational algebra representation. Next, this algebra representation is converted into a MAL plan, which is further optimized to derive an optimized plan. Finally the plan goes through an interpreted execution. Figure 3.1 displays such a plan for the following SQL query, from the TPC-H schema.

select l_tax from lineitem where l_partkey=1;

The plan is a sequence of semantically arranged instructions. Each instruction acts as an operator. The literals starting with "X_" are variables, which are assigned return values of statements. A statement has a syntax of the form "module.function(variable1, variable2,...)". For example, in the statement "algebra.leftjoin(X_23,X_10)", "leftjoin" is a function in the "algebra" module.

MAL comprises of a set of modules and a set of functions supported by each module.

MonetDB provides a GDB-like MAL debugger for fine grained analysis of operator execution at run time. However, further improvements can be gained by having a visual assistance tool, which also analyzes MAL execution traces. Stethoscope helps here by providing a set of functionalities to analyze MAL plans further, in a fast and efficient manner.

3.3.2 System architecture

The Stethoscope is a Java application and is built upon open-source products such as the GraphViz library [7], Zgrviewer component [21] of ZVTM tool-set [22], and the MonetDB profiler module [10]. Figure 3.2 shows the architecture. We describe each of these components in brief next.

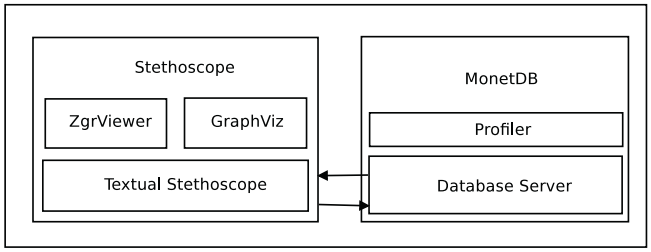

Figure 3.2: **Stethoscope architecture**

Database server is the MonetDB database server [10]. It is the main component which encapsulates the entire MonetDB execution environment. It works as a background process and listens for the incoming client connections on user defined ports. Stethoscope connects to Mserver as an ordinary client.

The profiler is a component in MonetDB kernel which profiles operator instruction executions. It supports profiling of events using several OS-specific properties, such as IO behavior, memory usage and cpu state, and MAL statement state. The profiler accepts filter options set through Stethoscope, which enables it to profile only a subset of event types. The events are either sent over a UDP stream back to the Stethoscope, or are dumped in a file, for offline analysis.

3.3.3 Graphviz

Graphviz is a graph generation library, and uses dot language to represent graphs. Upon request, the server generates a dot file for each plan before the execution begins. A dot file encodes a graph and describes the grammar for the representation of nodes, and the association between nodes and edges [62]. GraphViz can convert a dot file to a graph structure representation [7]. Stethoscope uses this graph structure representation to setup different navigational strategies.

3.3.4 ZGrviewer

ZGrviewer is an open source tool from the ZVTM tool set which provides interactive navigation functionality in a graph structure [21]. Its highlight is the zoomable interface which allows keyboard based and mouse scroll based navigation with zooming ability on individual nodes and edges in a graph. It has a plethora of features such as set of lenses viz. fish eye lens, etc. for visual interaction with graph nodes. ZGrviewer is implemented in Java and Stethoscope uses ZGrviewer interfaces for interactive navigation in the graph structure. Stethoscope code integrates with the ZGrviewer code base in a modular manner and provides interfaces for extensibility.

ZGrviewer stores graphics related meta-information in multiple structured object representations. Glyph is a structure representing a fundamental graphical object [118]. For example, consider a two node graph, with one undirected edge between them. Assume each node is represented with the shape of a circle and has a text label associated with it. ZGrviewer uses a glyph object each, to represent the shape, text, and edge. Thus for our example graph, it maintains the following objects, shape (two objects), text (two objects), and edge (one object). Other important objects are a virtual space, which represents a canvas on which graphs are dawn and a camera object, which shows different views at different zoom levels, in a virtual space.

3.3.5 Textual stethoscope

The profiler information is accessed through a textual version of Stethoscope. It uses a UDP socket interface to connect to the server, for receiving the execution trace. The textual Stethoscope can connect to multiple MonetDB servers at the same time to receive execution traces from all (distributed) sources. Its filter options allow for selective tracing of execution states on each of the connected servers. The profiler intercepted events on these servers are streamed back on a UDP connection to the textual Stethoscope. A sample execution trace from a trace file looks as given in the Figure 3.3.

3.3.6 Trace and dot file mapping

Each instruction is represented in the trace with two events. A “start” event marks the start of the instruction and a “done” event marks the end of the instruction. The program counter (pc) is an important field in the trace, and is used to map to a node number in a dot file. For example, an instruction execution trace statement with pc=1 maps to the node “n1” in the dot file. The “stmt” field in instruction execution trace represents a instruction and maps to the “label” field in the dot file.

3.4 Work-flow

The Stethoscope works in both online and offline mode. Both modes share some

```
[1] # event,  status,      thread,     pc,     totalticks,   stmt,          # name
[1] [ 0,      "start",     1,          0,      0,            "function user.s1_2 (A0=1);",   ]
[1] [ 1,      "start",     1,          1,      0,            "X_3 := sql.mvc();",        ]
[1] [ 2,      "done",      1,          1,      76,           "X_3 := sql.mvc();",        ]
[1] [ 3,      "start",     1,          2,      0,            "X_10:bat[:oid,:lng]  := sql.bind(X_3=0,\"sys\",\"lineitem\",\"l_tax\",0);",   ]
[1] [ 4,      "done",      1,          2,      92,           "X_10:bat[:oid,:lng]  := sql.bind(X_3=0,\"sys\",\"lineitem\",\"l_tax\",0);",   ]
[1] [ 5,      "start",     1,          3,      0,            "X_13 := calc.oid(0@0);",                   ]
[1] [ 6,      "done",      1,          3,      76,           "X_13 := calc.oid(0@0);",                   ]
[1] [ 7,      "start",     1,          4,      0,            "X_18 := A0=1;",            ]
[1] [ 8,      "done",      1,          4,      62,           "X_18 := A0=1;",            ]
[1] [ 9,      "start",     1,          5,      0,            "X_19:bat[:oid,:int]  := sql.bind(X_3=0,\"sys\",\"lineitem\",\"l_partkey\",0);",   ]
[1] [ 10,     "done",      1,          5,      74,           "X_19:bat[:oid,:int]  := sql.bind(X_3=0,\"sys\",\"lineitem\",\"l_partkey\",0);",   ]
[1] [ 11,     "start",     1,          6,      0,            "X_20 := algebra.thetauselect(X_19=  :bat[:oid,:int],X_18=1,\">\");",   ]
[1] [ 12,     "done",      1,          6,      63794,        "X_20 := algebra.thetauselect(X_19=  :bat[:oid,:int],X_18=1,\">\");",   ]
[1] [ 13,     "start",     1,          7,      0,            "X_22 := algebra.markT(X_20=nil,X_13=0@0);",        ]
[1] [ 14,     "done",      1,          7,      46,           "X_22 := algebra.markT(X_20=nil,X_13=0@0);",        ]
[1] [ 15,     "start",     1,          8,      0,            "X_23 := bat.reverse(X_22 );",          ]
[1] [ 16,     "done",      1,          8,      47,           "X_23 := bat.reverse(X_22 );",          ]
[1] [ 17,     "start",     1,          9,      0,            "X_24 := algebra.leftjoin(X_23=nil,X_10=  :bat[:oid,:lng]);",       ]
[1] [ 18,     "done",      1,          9,      173687,       "X_24 := algebra.leftjoin(X_23=nil,X_10=  :bat[:oid,:lng]);",       ]
[1] [ 19,     "start",     1,          10,     0,            "X_25 := sql.resultSet(1,1,X_24 );",          ]
[1] [ 20,     "done",      1,          10,     36,           "X_25 := sql.resultSet(1,1,X_24 );",          ]
[1] [ 21,     "start",     1,          11,     0,            "sql.rsColumn(X_25=1,\"sys.lineitem\",\"l_tax\",\"decimal\",15,2,X_24   );",   ]
[1] [ 22,     "done",      1,          11,     62,           "sql.rsColumn(X_25=1,\"sys.lineitem\",\"l_tax\",\"decimal\",15,2,X_24   );",   ]
[1] [ 23,     "start",     1,          12,     0,            "X_32 := io.stdout();",     ]
[1] [ 24,     "done",      1,          12,     34,           "X_32 := io.stdout();",     ]
[1] [ 25,     "start",     1,          13,     0,            "sql.exportResult(X_32==\"1399117138028 32\":streams,X_25=1);",  ]
```

Figure 3.3: **A Trace File**

fundamental steps, such as dot file parsing, conversion to an in memory graph representation, and sequential reading of a trace file. First, the dot file gets parsed and an intermediate scalar vector graphics (svg) representation gets created. In the next step, the svg file gets parsed and an in memory graph structure gets created. The root node of this graph structure is used for traversal in the graph at a later stage. Both steps use the Graphviz library interface. As a next step, Stethoscope parses the trace file in a sequential manner, storing attributes of the trace file. The "event" attribute from the trace is used as an index to store the attribute contents. The "pc" attribute is mapped to a node name, to search for the corresponding node in the graph structure, during graph traversal.

3.4.1 Offline execution analysis

It needs access to a preexisting dot file and trace file. Once the off-line mode is selected, and the initial dot file parsing to graph structure creation stage is over, interactive analysis begins. The system uses event based programming interfaces to monitor click events and takes appropriate action in response. Prominent actions are navigate to the next node in the graph, change color of a node, and display tool-tip text, etc. We describe the features related to these actions in the demonstration section.

During the off-line execution analysis, both dot and trace files have to be present in the same directory and should have the same name. For example, a valid path for dot and trace file is "/tmp/1.dot" and "/tmp/1.trace". A user can play with the following features, to analyze plan execution trace.

- Step by step walk through in graph nodes, analyzing individual instruction in trace using Stethoscope filter options window, debug options window, and tool-tip text display.

- Trace replay between two instruction execution states, with selective coloring on instruction execution time threshold.

- Jump to a specified execution state node in the trace.

- Birds eye view of the entire graph, to understand the instruction execution clustering, using coloring of nodes.

- Animation effects such as change of zoom level, variation in the transition time between highlights of two nodes.

3.4.2 Online execution analysis

Both dot and trace files are generated at run-time by the database server. Online mode components use a multi-threaded design. As a first step, the textual Stethoscope with all the filters options enabled is launched in a dedicated thread. The textual Stethoscope awaits in a listening mode for the arrival of trace stream on UDP connection. The trace received is redirected to a trace file.

The query whose execution plan needs to be analyzed is launched next in a separate thread. As the query execution begins, the profiler generates the instruction execution trace and sends to the textual Stethoscope. The monitoring thread filters the trace and a trace file is generated. The trace file continuously receives the trace stream from the textual Stethoscope, while the query execution is in progress. As the trace file grows in size, its content is sampled in a buffer. The query plans contain instructions where operator takes long time to execute, for example a join operator. When this occurs, the execution trace might block, resulting in blocking of the growth of the trace file.

The dot file is a prerequisite for the graph structure generation. The database server generates the dot file content and sends it over on the UDP stream to the textual Stethoscope, before query execution begins. A separate thread monitors the received UDP stream for dot file and execution trace file content. It filters the dot file content, generates a new dot file, and stores the content in it.

During query execution, the graph corresponding to query plan under execution gets displayed in the display window. A user can analyze plan execution, in the following ways.

- Monitor the progress of query plan execution using selective coloring of nodes in “RED” or “GREEN” color, based on algorithm described in section 3.4.3.

- Analyze and compare long running instructions using multiple instances of debug options window.

We describe the algorithm to color code execution analysis next.

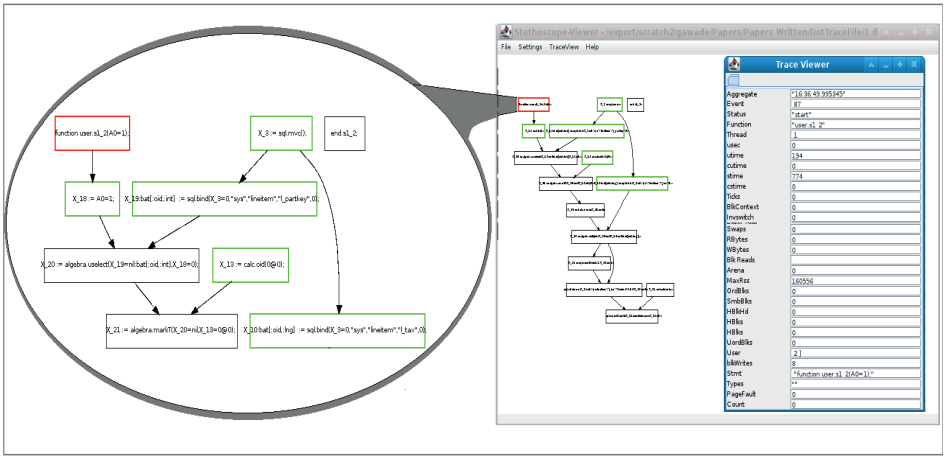

Figure 3.4: **A Display Window**

3.4.3 Run-time analysis algorithm

Finding long running instructions in a plan is one of the main purpose of the Stethoscope. Lengthy instructions can be filtered either on server or client side. We focus on the client side filtering. Lengthy instructions can be represented by color coding, progress window, and pop-ups, etc. We focus on coloring of nodes to represent state change.

Coloring graph nodes in an online stream is a complex task due to rendering limitations from the Java system. The Stethoscope uses the Java Event Dispatch thread queuing framework for queuing up nodes to render. This introduces a delay of up-to 150ms between rendering of consecutive nodes. A node is colored RED or GREEN based on the instruction status of “start” or “done” respectively.

Most instructions in the execution trace occur in sequence of pairs of “start” and “done” events. A consecutive “start” and “done” event status for the same instruction, with presence of more instructions afterwards, indicates that the instruction under analysis executed in least time. Hence, it is not a costly instruction. All such instructions are not colored. An instruction which does not appear in a sequence of pairs of “start” and “done” event is colored. For example, consider an execution trace buffer with fields such as {status,pc} representing 6 instruction statements {start,1},{done,1},{start,2},{done,2},{start,3},{start,4}. The graph nodes corresponding to first four statements will not be colored, as the two instructions corresponding to pc=1 and pc=2 occur in a pair, in a sequence. However, the graph node corresponding to the fifth instruction with pc=3 will be colored in RED. Thus this graph node coloring algorithm doesn’t check a specific instruction execution threshold time. We provide another algorithm which allows the user to specify an instruction execution threshold time.

3.5 Performance problem identification

Stethoscope helps in identifying various execution performance issues such as the execution skew in threads, blocking operators identification, top k expensive operator identification, identification of the operators that were executing in parallel with a specific operator to get the total resource consumption snapshot, operator scheduling issues, etc. We discuss a few of these in detail next.

Operator scheduling issues

The data flow graphs represent operator dependencies. Ideally an operator should start execution as soon as its inputs are available. The inputs arrive from the parent operators execution. Consider a simple example for a select operator represented by the instruction *X_3 = algebra.uselect(X_1, X_2);* During a visual analysis when X_1 and X_2 inputs are available, the nodes representing them turn GREEN. If both these nodes are GREEN, ideally the node representing X_3 should turn RED, indicating its execution is in progress, and then should turn GREEN to indicate the end of the execution. However, during some of the query execution we found that even though all input nodes were GREEN, some operators accepting these inputs did not turn RED for a long time. This indicates possible scheduling issues, which are very difficult to find otherwise in any manner. Run-time query execution visualization thus is immensely helpful in identifying performance critical issues to improve the system performance.

Execution skew identification

A visual inspection of the parallel graph representation shows distinct operator chains. For example consider query 8 from Figure A.3 in Appendix which shows four vertical dependency chains. The operators in these chains are executed by a single thread, such that 4 threads execute in parallel. By inspecting the start and end of the execution coloring in these chains, the execution skew is easily identifiable. The operator chain which takes longest to turn GREEN, shows skewed execution. Thus, the graphical representation of the multi-core execution indicates possible problems due to the execution skew.

3.5.1 Lessons learned

Considerable time was spent in integrating the code base of ZGrviewer with MonetDB trace, to apply the correct logic for coloring of individual graph objects, in online and offline mode. Rendering of nodes in tune with the online trace flow is a challenge, due to refresh rate related limitations of system. It required a lot of experimentation to get to a working state. We also encountered some synchronization issues during coloring and used locks to avoid it. A dot file parsing method from Graphviz tool set, which generates a simple graph structure as compared to the present one we use, was found to be faulty. User interface and event based programming in general involves tedious manual testing approach. Overall, we found the entire process a lot challenging than our initial anticipation suggested.

3.5.2 Future work

Stethoscope, as a platform provides interfaces to add new extensible features. Some of the main features planned are an analytic interface for micro analysis of trace information, gradient coloring of graph nodes to display a range of execution times based on their expensiveness, and selective pruning of plan to remove unimportant administrative instructions. Many trace analytic based features can be added easily.

An important observation is about the layout of the graph nodes. At present the layout is controlled by the graphviz graph generation algorithms, where the nodes could get an arrangement such that even though they indicate the data flow dependency, they might not follow the time order dependency. An example is, when one node executes in the top portion of the graph, while the next data flow dependent node executes in the bottom portion of the graph, while there are plenty of nodes laid out in between.

Thus Stethoscope falls short of identification of time ordered execution of operators. In the next Chapter we discuss our attempt to address this issue, by providing a visualization scheme that renders operator execution in a time ordered manner.

3.6 Applicability to other systems and related work

Performance analysis tools play a very important role in performance bottleneck identification. Most tools are designed with a system architecture specific focus and are used in different contexts such as troubleshooting problems in operating systems, database systems, compilers, etc. They offer functionality and features not necessarily needed in other contexts. For example, Intel Parallel Studio offers rich MPI based functionalities not necessarily needed in the database system context [35]. However, their generic usability principles and abstractions could be applied and tuned to benefit other systems. Visualization tools in parallelized query execution are critical as they allow parallelized execution bottlenecks to be identified easily, which otherwise is impossible using textual debugging tools. While Stethoscope's design favors MonetDB's plan representation, the concept of visualizing the plan execution is generic and can be applied to other columnar systems with appropriate modifications.

The HP-Vertica query analyzer [137] is a visualization platform that uses the Vertica column-store database with pipe-lined execution engine. In [137] authors use TPC-DS query execution in a clustered Vertica database engine. The Vertica query analyzer collects detailed thread level performance metrics for the operators of a running query. Some metrics are specific to the CPU usage, the memory usage, and the number of rows produced, while other metrics are operator specific such as network bytes sent per operator. The query analyzer uses principles similar to Stethoscope, in providing an interactive query execution analysis tool in a distributed database setting. Overall, it is inspired by Stethoscope.

Authors in [132] illustrate a 3D visualization tool for analyzing query execution plans in a clustered execution environment in SAP Hana in-memory database. Query execution 3D (QE3D) is a Java application based on a stand-alone architecture. It is installed on a client machine and processes interactive visualizations.

The usual process of performance analysis of different queries involves looking at various different diagram types in parallel, where each of them focuses on different aspects of a distributed query plan (e.g. network communication, physical operator interleaving, host utilization, temporal operator interleaving). Q3ED provides a holistic view of the entire query plan and enables analysis of different performance aspects in a single 3D space.

Another example is Vectorwise, a leading analytical database system which uses pipe-lined vectorized query execution. Vectorwise plans use a static graph representation with per node statistics, without any kind of interactive support. Integrating Stethoscope like interactive features in Vectorwise would enable interactive plan execution analysis for Vectorwise. One of the senior members of the Vectorwise team has investigated how Stethoscope could be integrated for Vectorwise. Other columnar systems can also benefit from interactive plan execution analysis inspired by Stethoscope.

3.7 Summary

The research question: In this Chapter we address question 1, "How well are the state-of-the-art database management system solutions using the available hardware resources?" and question 2, "How to provide insights into the query execution performance bottlenecks at a database system's functionality level?"

Identifying parallelized query execution performance bottlenecks is crucial to be able to improve parallelized query execution performance. Most database execution engines use text based analysis tools, which work reasonably well for small serial query execution plan analysis. However, parallel query plans tend to be much more complex. Hence, any assistance that expedites the process of parallelized query execution bottleneck identification is a crucial step in the research exploration.

Research contributions: We introduced Stethoscope a graph visualization tool to identify and analyze performance bottlenecks in parallelized query execution in a multi-core CPU environment. Graph visualization of the query execution trace combined with interactive analysis makes Stethoscope a valuable tool in troubleshooting query execution performance problems. Comparison of visualization tools is difficult as tools are custom built for individual systems. However, in a holistic sense, compared to the textual trace analysis tools which most database systems use, Stethoscope makes the performance bottleneck identification easier. This is one of the first of its kind tool available in the database research context, that allows both offline and online interactive analysis of query execution traces. Expedited identification of query execution performance problems is critical, hence such tools are a valuable research contribution.

We list important features that make Stethoscope a powerful tool, and highlight some use cases of performance troubleshooting to resolve them. The insights obtained using Stethoscope has led to crucial changes in the system architecture components such as the operator scheduler, the interpreter, different operator's implementation, etc. It has also inspired development of similar tools in commer-

cial systems such as Vectorwise and HP Vertica [137]. A strong interest to adapt Stethoscope to suit the Vectorwise query execution analysis tool was exhibited by a Vectorwise core team member.

3.8 Conclusion

Being able to pinpoint the performance problems in parallelized query execution is extremely crucial for performance improvements for any research related to parallelized query execution. Stethoscope is an extensible platform for parallelized query evaluation analysis. Developing a visual front end platform for analyzing textual execution trace for MonetDB, using open-source tools has been a challenging task. The principles of visualization are applicable to other columnar systems as well as could be seen from similar tools that were inspired by Stethoscope for other database systems. Its online and offline modes helps to identify and analyze some of the key query performance bottleneck issues, otherwise not possible using only textual trace analysis. Some of the prominent issues are identification of the expensive operators, execution skew in parallelized operators, and operator scheduling. Visualization based systems thus are a key component of the research ecosystem for multi-core parallelized query execution.

3.9 Sample TPC-H query data flow graph

In Figure 3.5 we list a query execution data flow graph to give a perspective of the complexity of the execution plan, when parallelized using static parallelization heuristic in MonetDB. A point to note is as the database system continuously evolves with better optimizer choices, efficient operator implementations etc., the plans become more compact, resulting in less complex graphs. The rectangles represent operators while the edges represent the data-flow. The aim here is to show the complexity of data plan in terms of its data-flow graph representation, without details about individual operators. Many of the operators are administrative operators, which have negligible cost, but need to be present for column store specific data flow dependencies.

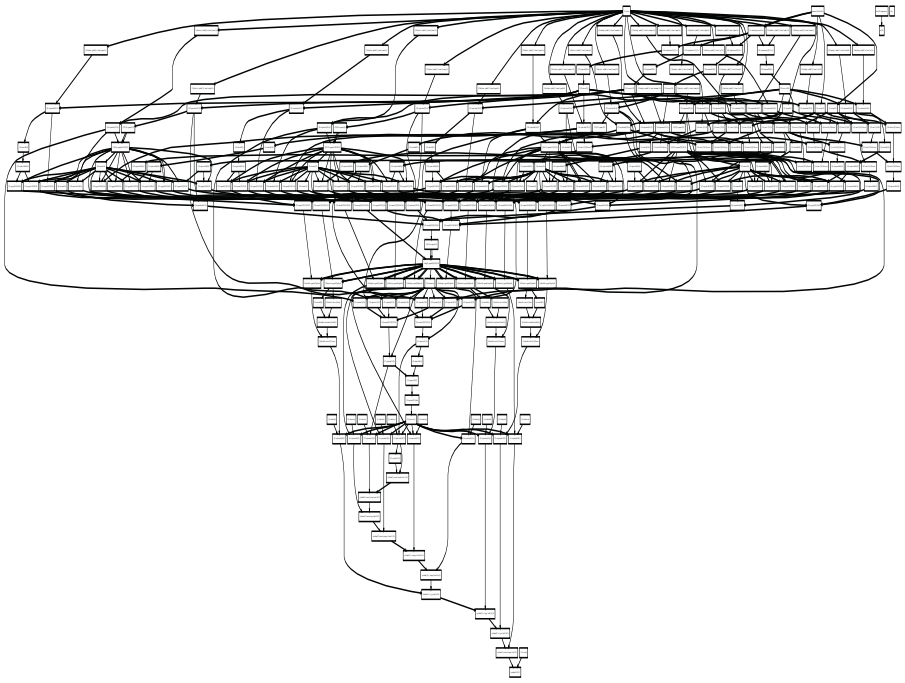

Figure 3.5: **Query 1, 220 nodes.**

Chapter 4

Query parallelization analysis through operator's execution order visualization

"'I do not seek. I find." — Pablo Picasso

Effective and controlled parallelization of database query plans on multi-core CPU's is a difficult problem. The problems arise due to the legacy software architectures designed for single core CPU systems, hardware architectural limitations such as limited memory bandwidth per core, lack of efficient scheduling models for multi-core CPU's, potential degree of parallelization problems, and inefficient heuristics about resource consumption and load balancing.

A fine grained analysis of where times goes during parallel query execution helps in identifying some of these issues. In this Chapter [1] we present "Tomograph", a tool to provide operator execution ordering based visualization, in an operator-at-a-time execution model columnar system. Tomograph visualization aligns operator execution with CPU, memory, and IO utilization in a multi-core environment. We classify and analyze the TPC-H query set, on the basis of query parallelization characteristics using Tomograph graphs. We identify issues in scheduling, partitioning, and resource utilization by operators, in parallelized query plans. As a solution, we discuss and experiment with the intra-operator parallelism using map-reduce multi-threaded approach. For the system under use, we find it less suitable than the inter-operator parallelism approach. We share the possible optimization opportunities for improvements and our learnings.

[1] This Chapter is based on the publication "Tomograph: highlighting query parallelism in a multi-core system", In Proceedings of DBTest, SIGMOD 2013.

4.1 Motivation

Query execution performance analysis tools assume a very important role for tuning database systems [5][135]. Some of the common methods for performance analysis are *explain plan* based operator statistics analysis [45][1], profiler based SQL query analysis [53], and query execution trace based fine grained analysis [11]. Explain based analysis often involves a visualization of query plans in a static graph format, annotated with operator statistics. Execution trace based analysis involves filtering of trace attributes and front end visualizers which aggregate trace information to provide a condensed view [87]. The volume of information contained in a trace, both in terms of number of attributes and quantity of information, could grow very large. The query performance analysis using the methods described before are well suited for SQL / operator statistics based analysis. However, parallel query execution analysis on the basis of operator time ordered issues is difficult using these methods.

Hence, operator execution ordering based visualization methods are critical in the world of query parallelization research ecosystem. During parallel query execution operator's execution ordering provides a lot of insight about the state of the system. Identifying problems such as computational skew in parallel threads is easy by visualizing the execution time-line of each thread side by side, as compared to deciphering it from statistics of individual operators in a graph visualization. Problems such as wrong operator scheduling in a complex plan which are very difficult to pinpoint otherwise can be spotted quickly by visualizing the operator execution ordering. Getting an insight about the possible parallelism problems is thus a matter of coming up with a correct visual scheme.

In this Chapter we introduce a new visual tool the *Tomograph*, that helps in identifying performance bottlenecks in a parallel query execution, on a single canvas. Tomograph provides a coherent operator's execution ordered view, cpu usage, memory usage, and disk IO activity. It improves the ability to pinpoint the performance issues during parallel query execution. We use it to explore the operator-at-a time execution model of the MonetDB execution engine [39].

4.2 Contributions

Text based tools do not give insights into the execution order of operators during query execution. We introduce a new tool, Tomograph, first of its kind, that visualizes the execution order of operators on a time-line, during query execution. Tomograph helps to identify where time goes at the granularity of an individual operator's level, during multi-core intra-query parallel plan execution. It also provides resource utilization information for the memory and the CPU. We briefly sketch the main contributions of this Chapter.

- Visual representation of parallelization behavior of operators in a query. Tomograph is the first tool to visualize operator's execution order during query execution.

```
function user.s1_1():void;
    X_2 := sql.mvc();
    X_3 := sql.bind(X_2,"sys","lineitem","l_tax",0);
    X_8 := sql.bind_dbat(X_2,"sys","lineitem",1);
    X_10 := bat.reverse(X_8);
    X_11 := algebra.kdifference(X_3,X_10);
    X_16 := sql.resultSet(1,1,X_11);
    sql.rsColumn(X_16,"sys.lineitem","l_tax",15,2,X_11);
    X_22 := io.stdout();
    sql.exportResult(X_22,X_16);
end s3_1;
```

Figure 4.1: **A serial plan in operator-at-a-time execution model.**

- Classification and analysis of TPC-H queries on the basis of parallelization characteristics. This is the first analysis of its kind in analytical systems.

- Identification of performance issues and possible solutions to resolve them, such as multi-threaded map-reduce based intra-operator parallelization.

4.2.1 Outline

The rest of the Chapter layout is as follows. In section 4.3 we provide a brief background of different existing query parallelization techniques. Section 4.5 focuses on architecture of Tomograph. Section 4.6 provides a detailed analysis of TPC-H benchmark queries. In section 4.7 we describe the experiments to improve parallelization in selected queries. Related work is described in Section 4.8. We provide a summary in section 4.9 and conclude in section 4.10.

4.3 Types of parallelization

Query parallelization can be categorized into two types, namely inter-query and intra-query parallelization.

Inter-query parallelization: Parallelization resulting from execution of two or more queries in parallel is termed as inter-query parallelization. All database systems when used in sequential / serial execution mode can provide inter-query parallelization, by executing one instance / thread on each core.

Intra-query parallelization: Intra-query parallelization is parallelism within a query. It can be further classified as inter-operator and intra-operator parallelization.

Inter operator parallelization is exhibited by two operators executing in parallel. Inter-operator parallelization is implemented using the exchange operator [68] about which we describe next. The *pipe-lined* query execution model also implicitly offers an inter-operator parallelization [70], as multiple operators execute in parallel during query execution.

Execution of a single operator in a multi-threaded set-up results into intra-operator parallelization. Most database systems do not offer intra-operator parallelization, due to difficulties in scaling issues, such as memory bandwidth limits.

Exchange operator: Most systems use the *exchange operator* for intra-query parallelization, as introduced by the *Volcano* system [68]. An exchange operator allows introducing parallelization in a query plan without modifying an operator's implementation. A query plan re-writer introduces multiple exchange operators in a query plan based on degree of parallelization needed. An exchange operator establishes a producer consumer relationship between itself and the original plan operator, which has now become its child. It forks a new thread / process for executing the child sub-plan. The total number of threads / processes forked control the degree of parallelism for the sub-plan. The data exchange between the two operators occur through a shared buffer / port [70]. Apart from the exchange operator based parallelism, the pipe-lined execution model itself offers an inherent parallel behavior. The parent operator consumes tuples as they are available from child operator. Hence, the blocking behavior is avoided.

4.3.1 Exchange operator in operator-at-a-time execution

In an operator-at-a-time execution model [41] an operator executes completely, materializing intermediate results. The operators in a query plan have a data flow dependency. This introduces a producer consumer relation between them. For example the serial plan in Figure 4.1 has operators following data flow dependency, where for example, instruction (X_10) can not execute until instruction X_10 has finished execution completely. However, not all operators follow the dependency pattern. Some operators are independent. The data flow dependency amongst operators establishes a serial order of execution on these operators. Only those operators can be executed in parallel which do not follow the data flow dependency. Establishing parallelization in operator-at-a-time execution model thus depends on the presence of data flow independent operators. Explicit parallelization is introduced in operator-at-a-time-execution model plans by introducing exchange operators in the plans, about which we describe next.

In our setup, operator-at-a-time execution operates on a decomposed storage model representation of data. The operators materialize the intermediate data completely. Plans are complex compared to traditional physical algebra representations, as they use one operator per column operation leading to too many operators, and also encode all operations such as tuple reconstruction using explicit operators. As the plans are represented in an abstract language, they undergo a series of optimizations (in compiler optimization style) using multiple optimizers. Parallelization is thus a result of applying a set of parallel plan generation optimizers called *Mitosis* and *Mergetable*, which introduce the *exchange operator* equivalent operators in the plan. Figure 4.1 shows a serial plan, whereas Figure 4.2 shows the corresponding partitioned plan generated by using Mitosis optimizer. In the previous Chapter we have illustrated how to read an operator-at-time-execution plan. The plan in Figure 4.1 corresponds to the following query.

```
function user.s1_2():void;
    X_2 := sql.mvc();
    X_3 := sql.bind(X_2,"sys","lineitem","l_tax",0);
    X_36 := sql.bind(X_2,"sys","lineitem","l_tax",0,0,8);
    X_39 := sql.bind(X_2,"sys","lineitem","l_tax",0,1,8);
    X_41 := sql.bind(X_2,"sys","lineitem","l_tax",0,2,8);
    X_44 := sql.bind(X_2,"sys","lineitem","l_tax",0,3,8);
    X_47 := sql.bind(X_2,"sys","lineitem","l_tax",0,4,8);
    X_50 := sql.bind(X_2,"sys","lineitem","l_tax",0,5,8);
    X_53 := sql.bind(X_2,"sys","lineitem","l_tax",0,6,8);
    X_56 := sql.bind(X_2,"sys","lineitem","l_tax",0,7,8);
    X_3:=mat.new(X_36,X_39,X_41,X_44,X_47,X_50,X_53,X_56);
    X_8 := sql.bind_dbat(X_2,"sys","lineitem",1);
    X_10 := bat.reverse(X_8);
    X_11 := algebra.kdifference(X_3,X_10);
    X_16 := sql.resultSet(1,1,X_11);
    sql.rsColumn(X_16,"sys.lineitem","l_tax",15,2,X_11);
    X_22 := io.stdout();
    sql.exportResult(X_22,X_16);
end s1_2;
```

Figure 4.2: **A range-partitioned query plan in operator-at-a-time execution model.**

select l_tax from lineitem.

In the rest of this section we describe the optimizer set which generates the parallel plans.

4.3.2 Mitosis

Mitosis partitions the base data on the largest table to generate an intermediate parallel plan. It uses an optimal serial plan as an input, and uses a heuristic using the available memory, the largest table size, and the number of cores to decide the number of data partitions. This design is influenced from the OLAP workloads in data warehouses, which use a star schema, with a large fact table and multiple dimension tables. The fact table is partitioned across multiple machines and dimension tables are stored locally on each machine. This helps in localizing parallel computations and avoiding large scale data transfers on the network. Mitosis uses a similar principle. One such sample partitioned plan is shows in Figure 4.2, where, *l_tax* column from the *lineitem* table is range partitioned into eight pieces. To maintain the original plan semantics across the plan, all the partitioned columns are joined together by a *mat.new* operator, which is similar to an *exchange.union* operator.

4.3.3 Mergetable

Mergetable identifies instructions in the partitioned plan with a dependency on the original instruction on which partitions were created. The original instruction in Figure 4.2 is *X_3 := sql.bind* and the dependent instruction is *X_11 := algebra.kdifference*. Mergetable expands the partitioned plan from Mitosis further by introducing new dependent instructions, to preserve the semantics of the original

```
function user.s1_2():void;
  X_2 := sql.mvc();
  X_3 := sql.bind(X_2,"sys","lineitem","l_tax",0);
  X_8  := sql.bind_dbat(X_2,"sys","lineitem",1);
  X_10 := bat.reverse(X_8);
  X_37 := sql.bind(X_2,"sys","lineitem","l_tax",0,0,8);
  X_58 := algebra.kdifference(X_37,X_10);
  X_40 := sql.bind(X_2,"sys","lineitem","l_tax",0,1,8);
  X_59 := algebra.kdifference(X_40,X_10);
  X_42 := sql.bind(X_2,"sys","lineitem","l_tax",0,2,8);
  X_60 := algebra.kdifference(X_42,X_10);
  X_45 := sql.bind(X_2,"sys","lineitem","l_tax",0,3,8);
  X_61 := algebra.kdifference(X_45,X_10);
  X_48 := sql.bind(X_2,"sys","lineitem","l_tax",0,4,8);
  X_62 := algebra.kdifference(X_48,X_10);
  X_51 := sql.bind(X_2,"sys","lineitem","l_tax",0,5,8);
  X_63 := algebra.kdifference(X_51,X_10);
  X_54 := sql.bind(X_2,"sys","lineitem","l_tax",0,6,8);
  X_64 := algebra.kdifference(X_54,X_10);
  X_57 := sql.bind(X_2,"sys","lineitem","l_tax",0,7,8);
  X_65 := algebra.kdifference(X_57,X_10);
  X_11:=mat.pack(X_58,X_59,X_60,X_61,X_62,X_63,X_64,X_65);
  X_16 := sql.resultSet(1,1,X_11);
  sql.rsColumn(X_16,"sys.lineitem","l_tax",15,2,X_11);
  X_22 := io.stdout();
  sql.exportResult(X_22,X_16);
 end s1_2;
```

Figure 4.3: **A range-partitioned query plan with operator dependency propagation, in operator-at-a-time execution model.**

serial plan. Thus, eight new *algebra.kdifference* instructions are introduced as seen in Figure 4.3. A *mat.pack* instruction finally combines work from all partitioned instructions. Mat.pack operator is similar to the *exchange.union* operator from the *exchange operator* family set of operators.

A data flow scheduler identifies the data flow independent instructions in the plan and schedules them for execution by the interpreter. If an instruction's input parameter is derived from an output variable of another instruction, dependency exists. As an example, consider instructions from the plan in Figure 4.3 with output variables (X_37,X_58), (X_40,X_59),(X_42,X_60),(X_45,X_61),(X_48,X_62), (X_51,X_63),(X_54,X_64),(X_57,X_65). Instructions in a pair are dependent. However, instructions across pairs are independent. Hence, they can be executed in parallel. Effective parallelism thus depends on identification of such dependencies and scheduling them efficiently. In a complex plan with many instructions, some times non-optimal scheduling decisions are taken. To some extent, *Stethoscope*, about which we described in the last Chapter, helps in identification of such problems [63].

4.4 How does operator's execution ordered visualization help?

Query parallelization helps to improve the response time of a query by dividing large computations amongst parallel processing units. Each query exhibits different

resource requirements. Analysis of individual operators from resource consumption perspective is often difficult as it gets affected by other operators executing in parallel. Better insight can be obtained by understanding a holistic system view, at any particular instance, where operators execution is ordered on a time scale. Operator-at-a-time execution model also offers better prospects for profiling and quantifying the performance of an individual operator, as it does not suffer from the profiling overheads associated with the tuple-at-a-time pipe-lined execution model. Next we briefly describe the visualization interface that orders operators execution on a time scale.

Visualization interface

Operator's execution ordered visualization provides a canvas to analyze the relation between operator execution, resource consumption and its effect on query performance. It is a consolidated graph which showcases the following.

1. Total number of active threads.
2. The start and end time of each operator.
3. The CPU activity per operator per thread.
4. The memory consumption by all operators.
5. IO consumption by all operators.

Figure 4.4 shows one such graph for TPC-H Q1. Each query operator is represented by a color. Execution of an operator is represented using a color box on a thread line. The length of the box indicates duration of operator's execution. The presence of a white space indicates no operator is under execution. In an ideal case there should be no white spaces in the graph during execution, indicating complete utilization of CPU cores. The CPU core activity is displayed in the top portion of the graph. Memory and IO utilization also get their own graphical representation.

4.5 Architecture

Tomograph is a command line client implemented in *C*, that connects to a MonetDB server (default branch change-set c56e636745dd), to receive profiled query execution trace. While the communication between server and client uses a UDP channel, the visualization is done using Gnuplot. Tomograph's work-flow can be split into the following stages.

1. Setting up the connection with the MonetDB server.
2. Collection and parsing of query execution trace data received from the server.
3. Preparation of the data for visualization.
4. Setting up Gnuplot based scripts to generate a consolidated graph.

On initialization Tomograph establishes a UDP connection with the MonetDB server and registers with the MonetDB profiler. The profiler starts when the query is launched. The default CPU profile interval of 50 milliseconds can be varied through Tomograph's *beat* option. While the CPU activity is sampled from the

/proc file system, the rest of the resources such as the memory and the disk IO are profiled at the *start* and the *end* of each operator's execution. Each statement in the query execution trace contains information such as the instruction counter, the instruction start / end state, the operator's name, the thread id, the elapsed time, and the memory and IO utilization. A query execution trace event is sent to the Tomograph, at the start and the end of each operator's execution. Since, CPU profiled events are generated at varying intervals, they are sent in a *ping* message header, on the same channel.

Tomograph listens for events from the MonetDB server in a loop and parses trace statements to retrieve information. The events are categorized in two types, the CPU events and the query events. The query event attributes are stored in an in memory data structure. When both start and end state of an operator is received, the association between an operator and it's visualization parameters such as placement, elapsed time, etc. is done. The CPU events data is stored in a separate file as number of events received can be very large. The listening continues till the query execution ends, or an interrupt is received from the user. Once the complete profiled data is available, the visualization phase begins.

Tomograph uses Gnuplot to generate multi-plots of profiled attribute information. The Gnuplot scripts are created based on received attribute information. The CPU core data is mapped to represent the CPU core activity, in an zero to one interval. The memory and the IO events similarly get their own representation. A box of varying length with a filled color is used to represent an operator's execution. The box length is determined on the basis of operator's execution interval. Most operators in an instruction have an execution interval, that is below the granularity of a box width display. Hence, only expensive operators gets a representation in graphs. A static color map is used to map operators to their colors.

4.5.1 Operator mapping

There are two types of operator in operator-at-a-time execution model. Relational algebra semantics operators (scan, join, aggregation, group, sort, and projection), and tuple reconstruction semantics operators. Administrative operators reconstruct the original tuple orders, from intermediate operations on individual columns.

The type of a query and the amount of partitioning has a direct effect on number of operators in a plan. However, most of these operators are inexpensive administrative operators. We classify the expensive operators next, and provide a mapping to their relational algebra semantics.

1. Select - algebra.uselect, algebra.thetauselect
2. Join - algebra.join, algebra.leftjoin, algebra.semijoin
3. Aggregation - aggr.sum. aggr.count, aggr.groupby, mat.pack
4. Arithmetic - batcalc.-, batcalc.+,batcalc.*
5. Groupby- group.multicolumn, group.sort

MonetDB stores base data and intermediate operation's data in binary association tables (bat). *algebra.leftjoin* is one of the most frequent operators in query

plans. It is used to do positional look-up on bats. For example, consider selection predicates *l_orderkey=o_partkey* and *l_returnkey= R* on TPC-H schema. Being a point select, if the second predicate gets evaluated first, it's returned results are stored in a bat. l_orderkey column would benefit from the reduced selectivity from l_returnkey results. Hence, a leftjoin operation is used on l_orderkey and returned results of *l_returnkey= R*. A join operator is then used for evaluating the results between results of leftjoin and the o_partkey column.

MonetDB uses highly optimized physical operators, implemented in "C". However, operators in a parallel setting affect each others behavior and performance gets less optimal. A visual overview by Tomograph of the time spent, provides deep insight into individual operator's behavior. In the next section we analyze individual TPC-H queries, to gain insights into their parallelization behavior.

4.6 TPC-H queries classification

Query parallelization is beneficial if the benefits outweigh the overheads of parallelization. Prominent overheads during query parallelization include the cost of partitioning and transfer of data, administrative and synchronization overhead of thread management, and the result merging cost. In a query that executes for a long time these overheads become negligible, however, a query that executes for a short time, the overheads stand out. Hence, parallelization does not benefit short running queries. TPC-H queries however, are a mix of long and short running queries. Having an insight into how long running TPC-H queries behave after parallelization helps to improve their performance. We classify the TPC-H queries on the basis of their degree of parallelization and analyze each query with respect to the following dimensions.

1. Degree of query parellelization.
2. Multi-core utilization by each operator.
3. Limitations of scheduling of operators.
4. Effect of blocking operators on query response time.
5. Intra-operator parallelism opportunities.
6. Limitations of operator implementations and partitioning policies.

The experimental setup is specified in Section 4.9. As degree of parallelization decides the number of CPU cores under use, it directly affects the multi-core utilization during query execution. We define multi-core utilization as follows.

Multi-core utilization: It is the sum of the execution time of all operators executed on all active threads, divided by the total time of execution from the start to the end for all active threads.

We provide a classification of selected TPC-H queries next using Tomograph's visualization scheme.

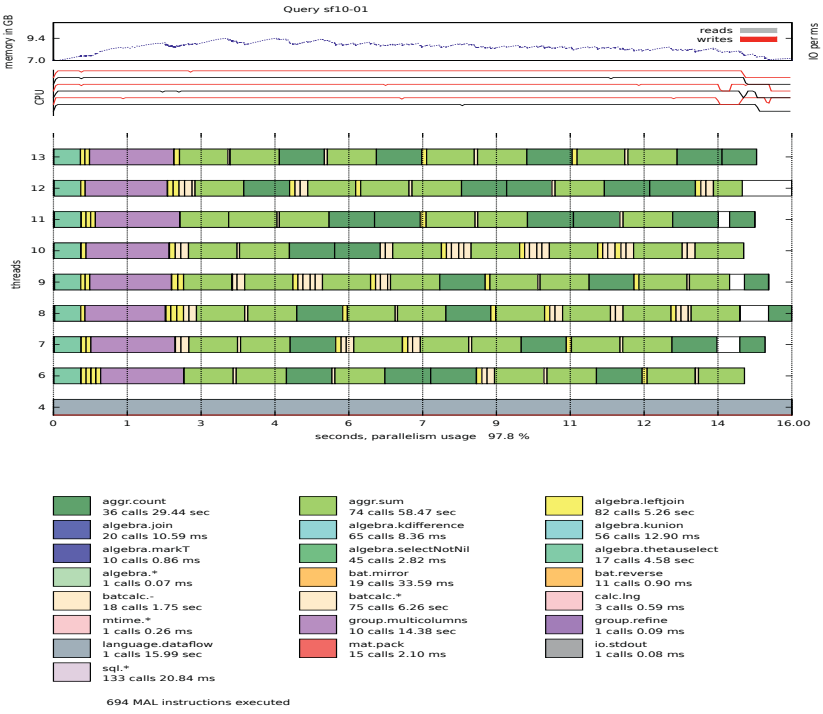

Figure 4.4: **TPC-H query 1 execution time-line on 10 GB data-set.**

Query 1: It is a simple query, which works on a single table *lineitem*. The most expensive operators are *algebra.thetauselect*, *group.multicolumns*, *algebra.leftjoin*, *aggr.sum*, and *aggr.count*. The query is dominated by aggregation and algebraic operators. At 97.5% the query shows extremely efficient multi-core utilization.

As *lineitem* is the only table present, it is partitioned. Parallelization of the plan is simple as there are no joins and the workers are evenly distributed. The prominent operations are group by and aggregations on the columns from the lineitem table. The selection operators work on the partitioned data, later followed by the group by operators, and further by the algebraic and the aggregation operators.

Overall, the query shows excellent parallelization, as it has a single large partitioned table and no complex operations like joins. As the intermediate data generated is large, the time spent in algebraic and aggregation operations is high and contributes maximally to the total query time. The multi-core utilization stays high due to computationally bound algebraic and aggregation operations.

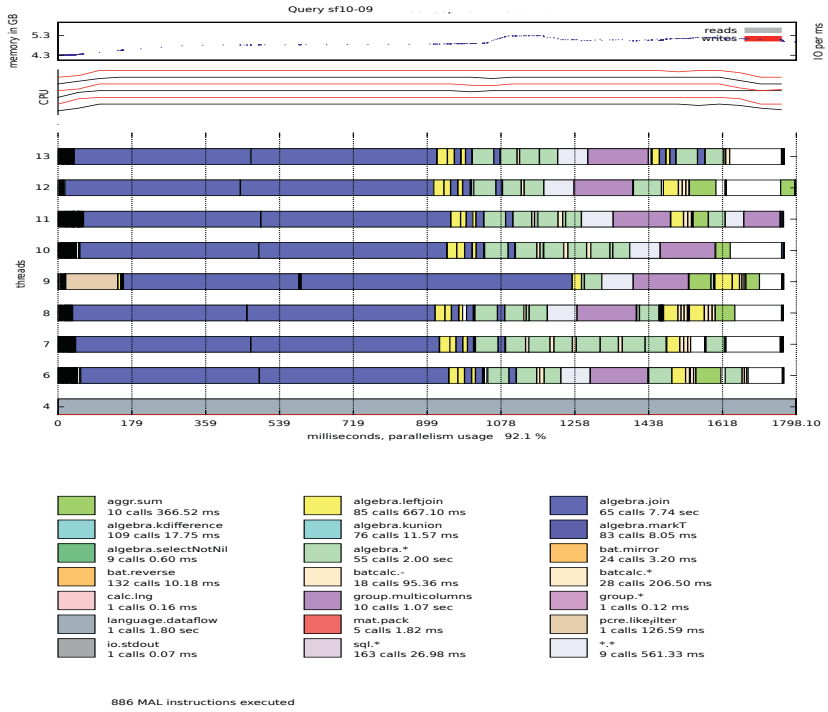

Figure 4.5: **TPC-H query 9 execution time-line on 10 GB data-set.**

Query 9: It is nested and more complex than query 1. Along with the lineitem table, it also has five other dimension tables. The most expensive operators are *algebra.join*, *algebra.letfjoin*, *group.multicolumns*, and *aggr.sum*. At 91.5% the query shows less multi-core utilization than query 1. This is evident from the presence of empty blocks towards the end of the query execution, before the group-by operator. Idle cores during execution mostly represent lack of available instructions, due to data flow dependent operator blocking.

Most joins in the inner query are present on the columns of the lineitem table. Hence, a partitioned lineitem table column is propagated in the plan to match partitioned join attribute column from other dimension tables. The inner query joins on lineitem table column's get evaluated first. Algebraic operations like subtraction, multiplication are done on some of the partitioned attributes from these tables on the inner query next. After a group-by on these results, the outer query does one aggregation before projection of results. Plan for this query is thus more complex than the first query due to presence of the joins and the inter-dependency of operations from the inner query to the outer query. However, the total data that is materialized from the joins in the inner query is relatively small. Also there are not many computational operations in projections to work on this data. Overall, the query has more memory bound operations like join than computational operations.

Figure 4.6: **TPC-H query 18 execution time-line on 10 GB data-set.**

Query 18: It is nested and more complex than the previous queries. Along with the presence of the lineitem table in both the inner and the outer query, there are two more dimension tables present. The most expensive operators are *group.done*, *aggr.sum*, *mat.pack (exchange union)*, and *algebra.join*. At 61.3% the query shows less multi-core utilization than the previous queries. Seven cores are idle when *aggr.sum* executes, making aggr.sum the blocking operator. There are instances of bad scheduling. For example. the second aggr.sum operator on the thread 12 should have been scheduled on the thread 10.

This query has a complex plan. The inner query has a group-by clause with an aggregation operation on the having clause. The columns involved are from the lineitem table, which provides an opportunity for partitioning. However, the plan becomes complex due to the tuple reconstruction phase due to the presence of group-by, having, aggregation and selection clause in a single predicate. The inner query also has two join clauses. They are present in the beginning on thread 8. This query can be distinctly divided into two regions. The region before, and after the blocking operator "aggr.sum", where parallelization is visible. The query spends more time on memory bound operations such as joins, than computational bound operators such as sum. The query also shows some unnecessary IO activity.

A complex plan thus offers less parallelization opportunities, as it has more data flow dependencies. Identifying blocking operators and exploring a possible intra-operator parallelization option on them is one possible approach to improve

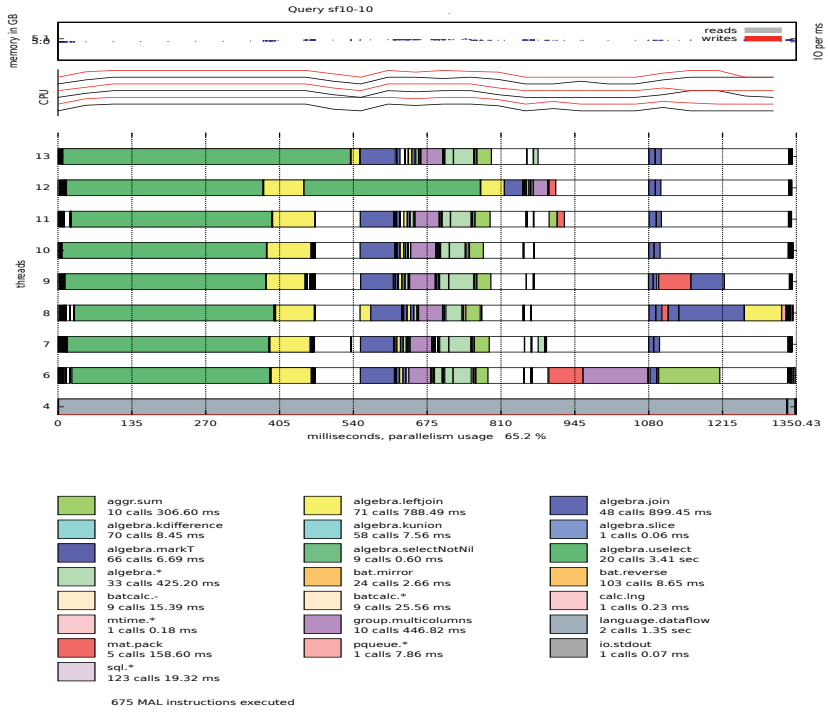

Figure 4.7: **TPC-H query 10 execution time-line on 10 GB data-set.**

parallelization as described in Section 4.7.4. "Aggr.sum" if partitioned can work on independent ranged partitions and could provide good parallelization opportunities. Being a computational operator it's efficiency depends on how fast it can access data from memory. Hence, memory bandwidth support during parallel execution could play an important role. Packing of results from different partitions could also be another bottleneck.

Query 10: It is relatively simple compared to query 18. Along with the lineitem table it also has three more dimension tables. The most expensive operators are *algebra.uselect*, *algebra.leftjoin*, *algebra.join*, *group.multicolumns*, *aggr.sum*, and *mat.pack*. At 64.9%, multi-core utilization is almost similar to the query 18. This is evident from white spaces during query execution. An analysis of the multi-core utilization offers deep insight into possible problems and future improvements. We discuss this later in "Incorrect scheduling and parallelization policies".

The query plan is relatively simple. The presence of a selection predicate on partitioned lineitem column triggers it's initial execution. Other selection predicates which involve a column from the lineitem table are data flow dependent on the previous selection predicate's result. Leftjoin (tuple reconstruction) operator works on the result of first selection predicate, and the lineitem table columns in the rest of the selection predicates for doing filtering. Join operations execute once leftjoin (tuple reconstruction) produces input for joins. The results are grouped by, and an aggregation operator is applied to find sum on lineitem table column. The

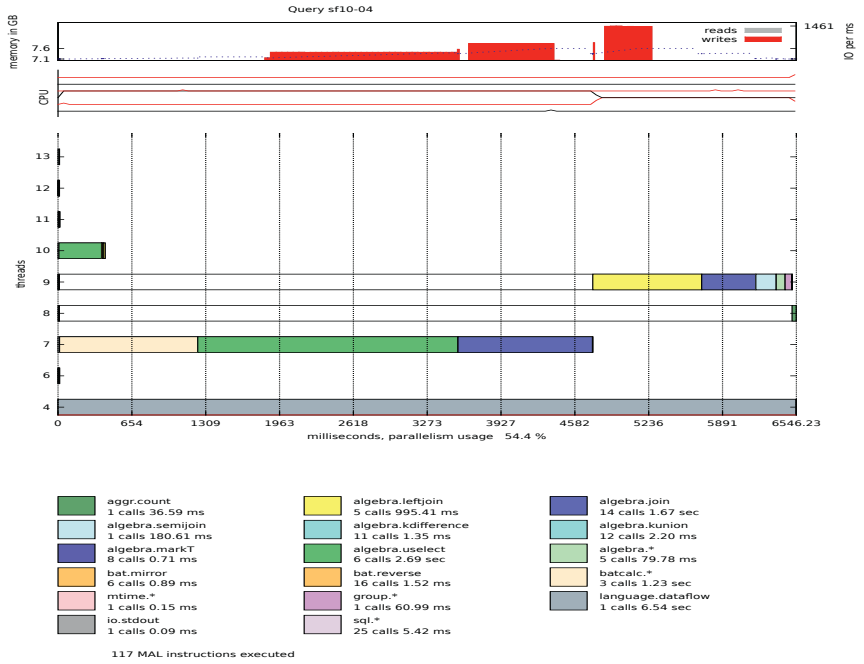

Figure 4.8: **TPC-H query 4 execution time-line on 10 GB data-set.**

query is blocked by multiple operators throughout execution. Select is one of the blocking operators.

Overall, the query shows good parallelization for most part of the execution. This can be attributed to the relatively simple plan which works on selection and join predicates on four tables in a partitioned manner. However, towards the end, the query shows blocking behavior. Whether it can be resolved by doing intra-operator parallelization of blocking operators needs to be explored.

Query 4: It is a simple nested query, whose multi-core utilization is 54.3% as the total number of active threads are too low. It shows the least parallelism, although the lineitem table is present. The reason being prevention of the plan parallelization as a precautionary measure to avoid too many join operators in the plan. Section 4.7.2 discusses more details about it. The query has a few expensive operators namely *batcalc.<(comparison)*, *algebra.uselect (selection)*, *algebra.join*, and *algebra.leftjoin (tuple reconstruction)*.

The outer query has a selection predicate on the orders table. Note that the orders table is not partitioned as the lineitem table is also present, and only the largest table gets partitioned by Mitosis. The “algebra.uselect” operator evaluating this predicate executes first, due to selection push down rule. The inner query has a join predicate and a selection predicate on the two columns of the lineitem table. The “batcalc.<” operator executes next, evaluating the comparison on the two lineitem table columns. “The “algebra.uselect” operator executes next, and it materializes the selection done by batcalc operator. Preparation for join predicate

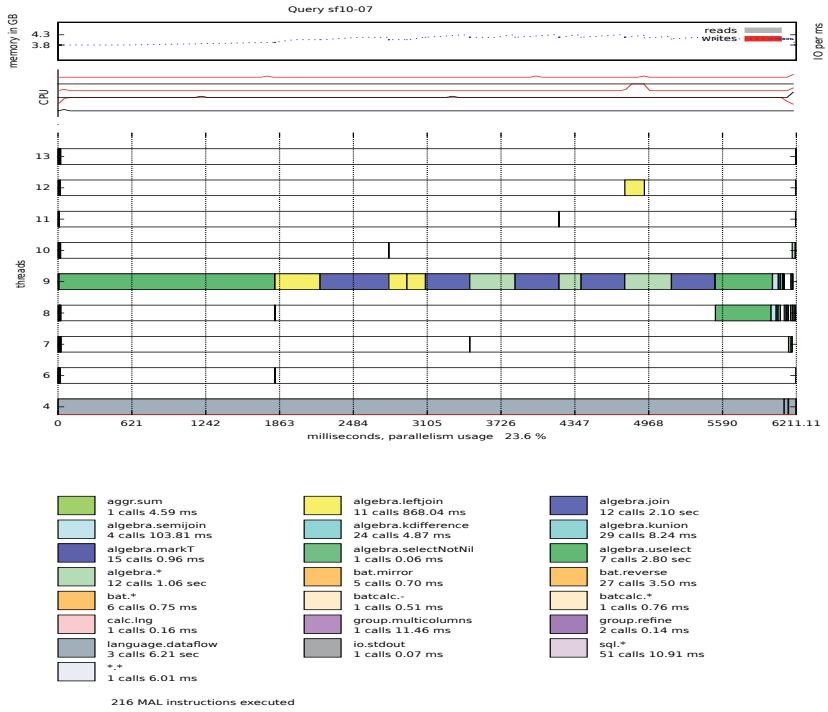

Figure 4.9: **TPC-H query 7 execution time-line on 10 GB data-set.**

evaluation happens next. It consists of an intermediate join between l_commitdate and l_receiptdate, the two columns in selection predicate. It gets represented on thread line after uselect operator execution. "Algebra.leftjoin" operation between the result of this join and l_orderkey happens next. The leftjoin does selection of values for l_orderkey. The main join predicate in where clause of inner query evaluates next. There is also an exist clause in the main query, which evaluates next. The "algebra.semijoin" operator represent execution of the exist clause. It's input is the result of the main join predicate, and the selection predicate on o_orderdata in main query. Groupby and count clauses evaluate next.

We expect that since the query involves computational operators such as batcalc and uselect, if parallelized, the speedup should be good. The query involves many intermediate joins even in the present form. These are used during intermediate stages, for tuple reconstruction. We discuss the case of "join" explosion further in section 5.1.2.

Query 7: It is a complex nested query which shows the least parallelism though it has the lineitem table. Like query 4, the optimizers do not parallelize the query plan, as it leads to the query plan explosion. The multi-core utilization is 23% as two threads stay active till the end. One thread is completely occupied, however the second thread gets to work only towards the end of the query execution. The query has a few expensive operators namely *algebra.uselect*, *algebra.leftjoin*, and *algebra.join*.

The inner query has multiple inter-dependent selection and join predicates. Join order resolution decisions are taken based on the predicate selectivity dependency. Due to selection push down rule, first a uselect on the lineitem column gets executed. Multiple join predicates in the rest of the predicates has a column from lineitem. Thus, they become dependent on the result of the first uselect. The dependency continues across columns from other tables which are used in join. The dependency chain continues till the last selection predicate on nation table gets resolved. This predicate gets executed in parallel as two separate independent uselect operations are involved.

A partitioning attempt on lineitem leads to a large plan expansion with expensive join operations. The plan explosion occurs due to the tuple reconstruction phase for intermediate operations, which are heavily inter-dependent. This prevents parallelizing the query.

4.7 Experiments

The experimentation platform is a machine equipped with Intel Core i7-2600 CPU @ 3.40GHz, 16 GB DDR3 Dual channel RAM, and 7200 RPM 1 TB SATA hard disk, running Fedora Core 16 operating system. MonetDB default branch changeset c56e636745dd is used. All the queries execute in memory on a TPC-H scale factor 10 GB data-set. Tomograph connects as a client to the MonetDB server. Queries are fired from a separate client to generate the graphs.

The experiments are intended to show that various performance bottleneck issues identified during visual classification of the TPC-H queries can be resolved using different parallelization solutions in the context of MonetDB. Some of these bottlenecks are as follows.

1. Limitations of operator scheduling.
2. Limitations of static data partitioning.
3. Limitations of static heuristic.
4. Blocking operators.

A detailed analysis of some of the queries, on the above factors and possible solutions for their improvement are discussed next.

4.7.1 Scheduling and partitioning policy limitations

Q10 in Figure 4.7 is a good example of problems in operator scheduling and data partitioning. The query has one scan select operation on the lineitem and the orders table column (black). Since MonetDB parallel plan optimizer always partitions columns in the *largest* table, the presence of 8 CPU cores leads to 8 equi-range disjoint partitions of the lineitem table column. While one *select* operator works on each partition, a single *select* operator works on the orders table column, since it is not partitioned.

With the 8 execution threads, the scheduler has two possible scheduling choices. First, to schedule 8 select operators on the 8 way partitioned column, or to schedule 1 select operator on the non-partitioned single column and 7 select operators

on 8 way partitioned column. We observe that the scheduler uses the 2nd choice by scheduling a single select operator (topmost left black) on thread 13 and 7 partitioned select operators (black) on threads 6 to 12. The 8th select operator gets scheduled on thread 12 after the first select operator scheduled on it finishes its execution. While the select operators on threads 6 to 11 finish first, they continue to wait until the select operation on the non-partitioned column (thread 13) finishes. This wait introduces idle time on multi-cores. The partitioned join operator which is scheduled next (purple) needs results from both types of (non-partitioned column and partitioned column) select operators as its input. The join operators following the 6 select operators (thread 6 to 11) start execution as soon as the select operator on thread 13 finishes.

Solution

This case shows the problem of incorrect scheduling order and incorrect number of partitions. One possibility is to parallelize the select operation on the single non-partitioned column of the orders table and schedule it to execute first. This would serialize the execution of select operations on the orders and the lineitem table, thereby removing the waiting time.

The other possible resolution is by identifying the correct number of partitions. For example, instead of 8 partitions of the lineitem table column, use only 7 partitions. This would avoid waiting time for other threads, as they wait for the 8th select operation to finish.

Identifying exact data partitioning range such that optimal load balancing is achieved is a difficult problem, as resource consumption changes dynamically. In a simple case, if the lineitem table partitions are reduced from eight to seven, each of the seven select operators get more range of data to work on. Hence, their execution time increases. This could keep the seven threads busy till the select operator on thread 6 finishes.

However, the decision to reduce number of partitions dynamically is difficult, as it involves run time monitoring and synchronization amongst all executing operators and needs a continuous load balancing strategy. The plan generation at present is static and dynamic monitoring and adaptation infrastructure does not exist. However, we identify this as a possible future research direction to solve such type of problems.

As MonetDB does not use a cost based optimizer, an important question is, if a cost based optimizer could have identified the correct number of partitions. In Chapter 6 we discuss the role of correct number of partitions in plan parallelization. We also checked the query plan for Vectorwise, a leading analytic system, since it uses a cost based optimizer. It uses both the lineitem and the orders table in a partitioned manner. It is able to partition the tables correctly.

4.7.2 Static heuristic limitations

The problem of queries involving the lineitem table not getting parallelized, as seen in the query 4 case (See Figure 4.8), compelled us to investigate such type of

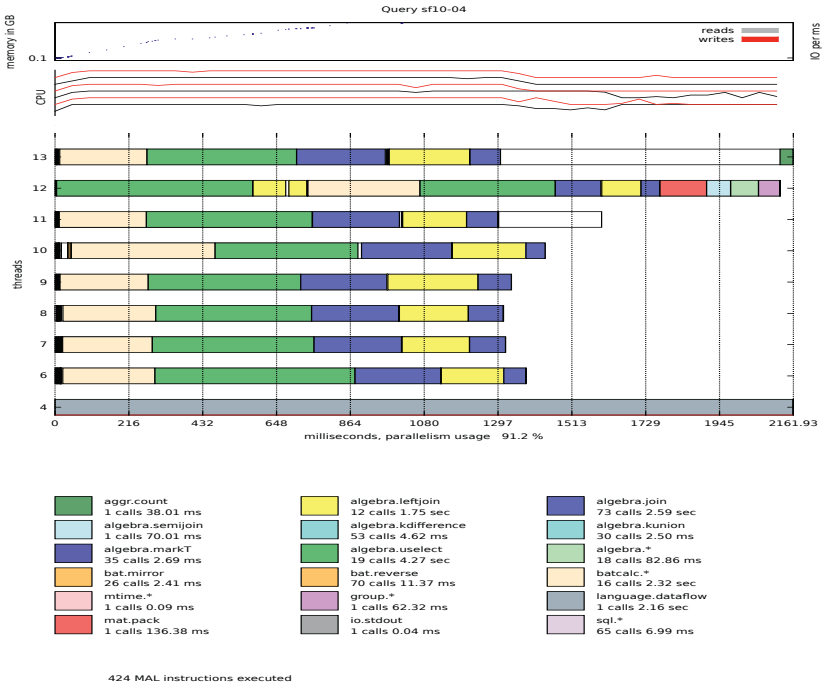

Figure 4.10: **TPC-H query 4, forced expanded parallel plan execution on 10 GB data-set. The timing is improved by 3 times the timing of query 4 without forced plan expansion.**

queries. Query 4 does not get parallelized due to the problem of "join" explosion in a large plan expansion. Based on the properties of the data (unique, random) the amount of work done by "join" operator varies. If the optimizer detects a possible cross product during join, then the total complexity of the join calculation increases, due to too many joins. The heuristic in parallel plan generation optimizer checks for such conditions, and if matching conditions are found, it rolls back the expanded plan to a sequential plan.

Solution

We do an experiment to check the severity of the join explosion problem by forcing a plan expansion for the query 4. The heuristic condition in the plan re-writer that checks for the "join" explosion condition is deactivated. The resultant expanded plan contains 424 instructions with 60 new join operators, while for the non-expanded plan there are 117 instructions. The expanded plan however, exhibits very good parallelism as seen in Figure 4.10, with the query execution improved from 6 seconds to 2 seconds, i.e. by three times. Execution time of individual operators also decreases by around 4 times. This case shows that static heuristics in parallelization decisions can be sub-optimal.

4.7.3 Blocking operators

Blocking operators is one of the most prominent visible bottleneck. They arise due to data flow dependency amongst operators in operator-at-a-time execution model, where until all the input parameters are available an operator can not start it's execution. A "sort" operator in relational algebra is a blocking operator, as sorting requires availability of the entire range of input.

However in MonetDB context, depending on the other operators under execution and resource availability normal operators could show blocking behavior. A broad level categorization is based on partial blocking or complete blocking behavior. The select operator on the orders table predicate in query 10 in Figure 4.7 (Thread 13) shows a partial blocking behavior. While its under execution 7 threads on the lineitem table column execute select operators, until the join operator (purple) dependency arrives. In contrast, when the operator "aggr.sum" in query 18 (Figure 4.6) executes, it blocks all operations, showing a complete blocking behavior. There are non-blocking partitioned sum operators in the same query. The complete blocking operator behavior results, as the blocking operator receives all the data flow dependency operators as its input.

Solution

Blocking operators if parallelized in an intra-operator manner, could improve the query response time. "algebra.uselect" and "aggr.sum" show blocking behavior in many query plans. We focus on "algebra.uselect" as it occurs more frequently in different contexts. In the next experiment we describe its intra-operator parallelization.

4.7.4 Parallelization of blocking operators

Operator parallelization uses either inter-operator or intra-operator parallelization. In both the input is range partitioned and selectivity condition is applied on each partition. Inter-operator parallelization has multiple operators operating in parallel. MonetDB supports inter-operator parallelization by default, with a single threaded operator execution.

On the contrary, intra-operator parallelization uses a multi-threaded implementation of an operator. An operator could have multiple implementations to satisfy different cases, for example, MonetDB uses variants of the operator *algebra.uselect*. An optimal implementation is chosen on the basis of properties of data such as sortedness, uniqueness, etc. However, the intra-operator operator parallelization does not benefit all of them. Parallelization of uselect makes sense only in those cases where sufficient computational opportunities are present. Since range selection over random data is one such case, we focus on the intra-operator parallelization of the select operator, for this case.

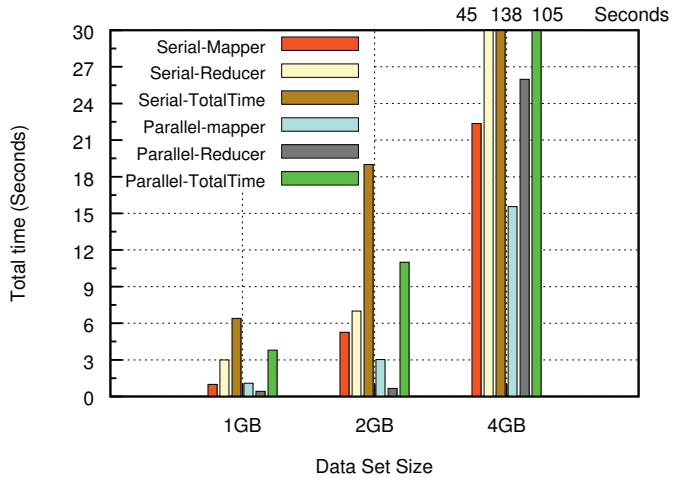

Figure 4.11: **The comparison of Intra-operator uselect, with serial and parallel reducer phase.**

Intra-operator parallelization

"Algebra.uselect" operator is parallelized using a multi-threaded map-reduce approach [125][105]. Mapper functions are forked on multiple threads to do range based selection. A multi-threaded reducer phase combines these results. The reducer produces the final buffered results by combining data from different mappers either in a serial or parallel manner. Thus, based on the type of reducer used, we categorize the intra-operator parallelization of uselect in two types. The intra-operator serial version and the intra-operator parallel version. The total time in both version can be divided as below.

1. Forking of mapper and reducer threads.
2. Computational work by mapper threads.
3. Buffer copy by reducer threads.

To understand the time division in intra-operator parallel phases, we conduct a simple experiment. A simple query ***"select count(*) from table where a >1"*** is used on an in memory cached server. The table contains random data of a single attribute of type "long". Data sets of size 1GB, 2GB, and 4GB are used. Selectivity of the query is 100%. Both the serial and parallel versions of intra-operator uselect are compared. The graph in Figure 4.11 displays the time division for the mapper phase, reducer phase, and total time for the query. Following observations are made.

1. Mapper threads take less time than reducer threads.
2. Serial version reducer takes more time than parallel version reducer.
3. The proportion of total mappers and reducers time, to the total query response time is more in serial version, than in parallel version.

4. The query response time does not increase in proportion to the data-set size increase.

Mappers: Eight threads are forked in each mapper and reducer stage. Thread forking is a lightweight activity and takes around twenty microseconds per thread. To save thread forking costs, another alternative is to use a thread pool. Mapper's spend their time in computation and memory access, where the computation in the current case is of comparison type. Since memory access is always slower than CPU clock speed [27], the time spent in reading and writing the data is memory bandwidth dependent. It varies based on resource contention amongst competing threads. Hence, the increase in time for mappers across different data sets, is not in proportion with the increase in data set size. Memory access optimization could thus improve mapper performance further. This is further discussed in Section 4.7.5 in memory bandwidth measurement.

Reducers: Merging results from multiple mappers to a final buffer in the reducer phase is always a bottleneck. Copying time is spent in reading various buffers from different memory addresses, and writing data at a separate location. Depending on the size of the data to be copied, the timing of copy could vary considerably. Buffer copying is expensive if done in a serial manner. Each reducer thread copies data serially to the destination buffer. Parallel copying reduces the cost significantly, as multiple threads copy simultaneously.

For small data sets sized 1GB and 2GB, the parallel version is an order of magnitude faster than the serial version. However, for the 4GB data set the query execution time is 45 sec in serial, and 26 sec in parallel version. This is attributed to the disk IO effect and operating system noise. MonetDB allocates intermediate buffers (bat), using heap or a memory map based allocation. For small data size up-to a few hundreds of MB's, bat's use a heap allocation. For larger size, bat's use a memory mapped allocation. Memory mapped allocation could trigger a disk IO, based on the memory pressure. The operating system with its background process activity could also trigger flushing of memory. We suspect these two factors leads to the bad performance of intra-operator parallel version.

The proportion of all mappers and reducers time, in comparison to the entire query execution time, is an important metrics. Apart from "uselect" other important operators in the query are "leftjoin", and "sum". In the serial version, the combined time of mappers and reducers is 60% of the total query time. In parallel version this proportion is just 40%. The overhead of other instructions in parallel version emphasizes the effectiveness of uselect parallelization. It calls for an efficient implementation of other instructions as well.

Query 4 uselect intra-operator parallelization

Query 4 is another good candidate to test intra-operator parallelization of the "uselect" operator. Out of it's six uselect operators, only two are expensive and contribute up-to 2.65 seconds. The experiment is intended to observe the effect of

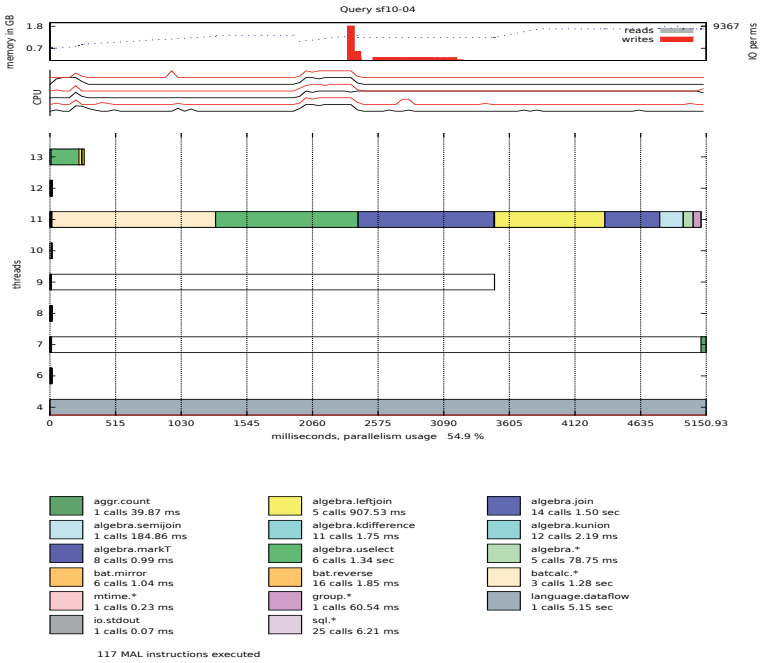

Figure 4.12: **Query 4 with intra-operator uselect parallel version. The uselect operator time has improved by two times, as compared to uselect operator time in query 4 without intra-operator uselect.**

intra-operator parallel uselect, on improving the response time of a TPC-H query. The graph in Figure 4.12 shows the result. The total time for uselect is reduced to 1.34sec from 2.65 sec. This is an improvement by two times. We expect an improvement of at least up-to four times, due to the presence of four physical cores. The time for mapper and reducer phases are stable and together contribute around 700 ms. We suspect memory bandwidth can be an issue, hence measure it next.

4.7.5 Memory bandwidth measurement

Parallelization of operators such as select also depends on the memory bandwidth, as too many operators accessing memory in parallel puts pressure on the memory. The Intel i7 core 2600 quad-core processor has a peak bandwidth of 21GB /sec. The graph in Figure 4.13 plots the read / write memory bandwidth experiment results. We measure the effective bandwidth of a single, two, four, and eight threads. Some of the important observations are.

1. Maximum read bandwidth is 17.39GB/sec.
2. Maximum write bandwidth is 8.5GB/sec.
3. Read bandwidth for four and eight threads is similar.

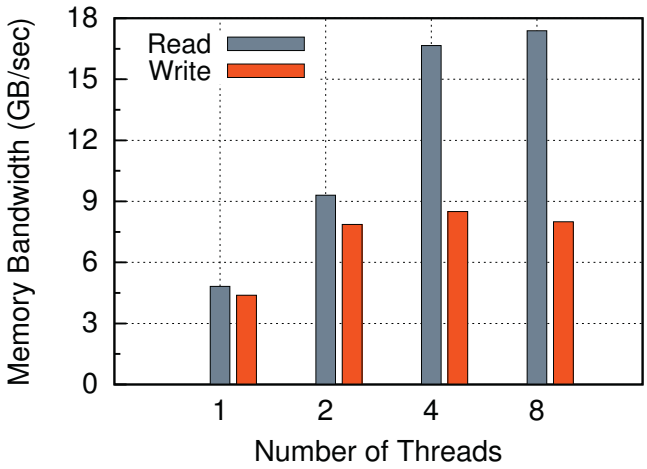

Figure 4.13: **Read / Write memory bandwidth comparison.**

4. Write bandwidth for two, four, eight threads is similar.

The experiment uses GCC 4.6.3 compiler. An array variable of type *int* is allocated 4GB memory and initialized. In order to measure the read bandwidth, loop unrolling for 16 times is done to compute "sum = array[i] + 1". At the end all individual "sum" are added to nullify the effect of dead code elimination by the compiler. The write bandwidth is measured using the expression "array[i]=sum". The experiment is conducted in both single thread and up-to 8 threads case. In the multi-threaded setup each thread works on a range of the array. The experiment is first carried without any compiler optimizations, and then repeated with the *-o1, -o2, and -o3* optimizations. However, no noticeable differences are observed in the execution times.

Hyper-threading provides a logical core along with an existing physical core. In Figure 4.13 4 threads represent physical core threads, whereas 8 threads includes 4 hyper-threads. It benefits the thread in using the idle CPU execution units, when the thread executing on the physical core is busy. However, as shown in Figure 4.13, it has no effect on the memory bandwidth. This explains the reason both four and eight threads show similar read / write bandwidth. Read experiment bandwidth for four/ eight threads is around four times better than the single thread bandwidth, and is close to the theoretical bandwidth of 21 GB/sec. At 8.5GB/sec write bandwidth is around half of the read bandwidth. Hence, writes are expensive and any optimization in saving writes in mapper phase should improve mapper's performance. Write bandwidth stays almost constant after two threads. We are unable to explain why write bandwidth is so low.

We have given an overview of different performance bottleneck issues during parallelized query execution using time ordered visualization for selected TPC-H queries. The visualization scheme proposed by us has helped us identify these issues. Tomograph is the first tool in the database system context to propose such a

visualization scheme for time ordered analysis of query execution.

4.8 Related work and applicability to other systems

Most systems use the "Explain" command for parallel query plan visualization. Other auxiliary tools also use the explain semantics with added functionality, such as operator node statistics and color coding [5]. Microsoft SQL server uses a suit of tools titled "SQL server management studio" [18]. Vectorwise uses a graphviz generated static visualization plan tree, with color coding. Postgres uses similar tools [17]. However, none of these tools provide a visual parallel execution overview of operators, in a time ordered manner, in a multi-core setting. Problems in parallelization of query plans are detected by manual analysis of statistics provided in each operator. Often color coding is used to highlight problematic operators in a plan. For example, Vectorwise uses gradients of a single color to paint nodes in an execution plan tree. In a parallel plan, dark gradient color represents a skewed node, whereas a normal node gets light color. Further analysis is done using statistics from each node.

The closest tool similar to Tomograph in the context of database system is Q3ED, [132], which is a 3D visualization tool for analyzing query execution plans in a clustered execution environment in SAP Hana in-memory database. Query execution 3D (Q3ED) is a Java application based on a stand-alone architecture. It is installed on a client machine and processes interactive visualizations. The usual process of performance analysis of different queries is looking at various different diagram types in parallel, where each of them focuses on different aspects of a distributed query plan (e.g. network communication, physical operator interleaving, host utilization, temporal operator interleaving). Q3ED provides a holistic view of the entire query plan and enables to analyze different performance aspect in a single 3D space. It is inspired by the Stethoscope and Tomograph tools.

Intel Vtune analyzer is another tool that allows visualization of any process under execution, hence could be used also for visualization of database execution engine process. It provides different generic visualization themes such as the part of the code where most of the time is spent, etc., however, none of these schemes are specifically designed with a focus on the database query execution unlike Tomograph.

A direct comparison with other such tools can not be done because lack of such tools, and even if they exist they are specific to the system under use. Tools are always used to assist in the progress of the subject research area, and should help improve the overall research ecosystem by providing better abstraction and generic principles that could be utilized in other systems, rather than from the perspective of their absolute effectiveness in comparison with similar tools.

4.9 Summary

The research question: In this Chapter we address the questions, ”How well are the state-of-the-art database management system solutions using the available hardware resources?” and ”How to provide insights into the query execution performance bottlenecks at a database system’s functionality level?”

Query execution bottlenecks result from different factors such as operator scheduling problems, operator implementation problems, run-time resource contentions, etc. The user interface of the text based performance analysis tools is not sufficient to identify problems arising due to these factors. Identifying parallelized query execution performance bottlenecks is crucial to be able to improve their performance. Hence, any assistance that expedites the process is a crucial step in the research exploration. In this Chapter we have investigated how visualizing operator’s execution order on a time-line can assist in identification of query execution performance bottlenecks.

Research contributions: We introduced Tomograph, a visualization tool to identify and analyze the performance bottlenecks in parallelized query execution in a multi-core CPU environment. Visualization of execution order of operators on a time-line allows easy identification of crucial bottleneck problems such as operator scheduling, multi-core utilization, hence it acts as a valuable tool in troubleshooting query execution performance problems. We have analyzed a selected set of TPC-H benchmark queries for different types of performance bottleneck problems. This kind of visual analysis is done for the first time in the database world, and we provide different use cases to show its effectiveness.

A relative comparison between tools is hampered by their context. They are custom built. Closest comparison could be with VTune analyzer [126], a profiler and visualization tool from Intel for generic process performance issue identification. Tomograph is one of the first of its kind tool catered specifically towards multi-core database systems. Its generic visualization principles are applicable to be used in other systems as well. Expedited identification of query execution performance problems during research explorations are critically dependent on such tools, hence such tools are a valuable research contribution.

Stethoscope, the visualization tool we described in Chapter 3, does not provide insights into an operator’s execution order dependencies with respect to time. Tomograph’s visualization approach was inspired by this deficiency. Visualizing an operator’s execution dependencies with resource utilization (CPU core utilization and memory utilization) with respect to time provides crucial insights into the run-time behavior of the system. It helps to analyze the run-time issues during an operator’s execution with respect to scheduling, robustness, degree of parallelization and multi-core utilization, memory utilization, etc. Tomograph has inspired the development of similar tools in commercial systems such as SAP Hana [132] and HP Vertica [137].

4.10 Conclusion

Tomograph is a new tool in the database research world that provides a visual approach to identify problems in query parallelization in a multi-core setting. It helps to understand and analyze where time goes during operator execution in a multi-core query execution setting. We classify and analyze the TPC-H query set on the degree of query parallelization and multi-core utilization. We classify the possible problems hampering parallelization as limitations of scheduling and partitioning decisions, limitations of the static heuristic rules, and blocking operators. Solutions to these bottleneck issues are proposed. Tomograph has inspired similar visualization approaches in other database systems.

Chapter 5

Adaptive query parallelization in multi-core column stores

"Change is the end result of all true learning." - Leo Buscaglia

With the rise of multi-core CPU platforms, their optimal utilization for in-memory OLAP workloads using column store databases has become one of the biggest challenges. Some of the inherent limitations in the achievable query parallelism are due to the degree of parallelism dependency on the data skew, the overheads incurred by thread coordination, and the hardware resource limits. Finding the right balance between the degree of parallelism and the multi-core utilization is even more trickier. It makes parallel plan generation using traditional query optimizers a complex task.

In this Chapter [1] we introduce *adaptive parallelization*, which exploits execution feedback to gradually increase the level of parallelism until we reach a sweet-spot. After each query has been executed, we replace an expensive operator (or a sequence) by a faster parallel version, i.e. the query plan is morphed into a faster one. A convergence algorithm is designed to reach the optimum as quick as possible.

The approach is evaluated against a full-fledged column-store using micro-benchmarks and a subset of the TPC-H and TPC-DS queries. It confirms the feasibility of the design and proofs to be competitive against a statically optimized heuristic plan generator. Adaptively parallelized plans show optimal multi-core utilization and up to five times improvement compared to heuristically parallelized plans on the workload under evaluation.

[1] This Chapter is based on the publication "Adaptive query parallelization in multi-core column stores", In Proceedings of EDBT 2016.

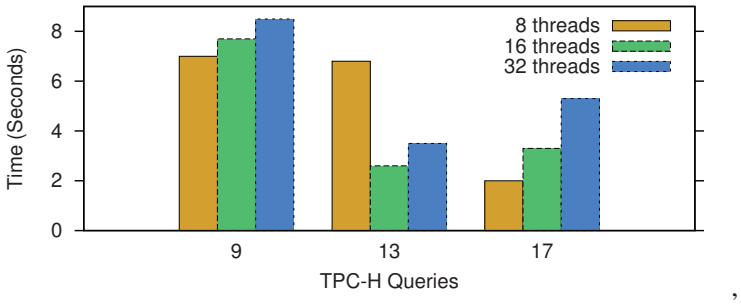

Figure 5.1: **Response time variations due to varying degree of parallelism under concurrent workload (32 hyper-threaded cores).**

5.1 Motivation

Column store databases are designed with a focus on analytical workloads. Almost all database vendors these days have a column store implementation. A recent study by Microsoft showed that a majority of real world analytic jobs process less than 100 GB of input [31]. This can be accommodated by an in-memory solution on a single high-end server. They come with an abundance of CPU power using tens of cores [130, 113]. Query parallelization is one of the ways to utilize multi-cores. This calls for a renewed look at the traditional query parallelization techniques, such as the *exchange operator* based parallelization [68], since the state-of-the-art column store systems such as IBM BLU accelerator [124], HyPer [142] use work stealing based approach for multi-core scalability.

An important issue is the degree of parallelism **(DOP)** of a plan which reflects the maximum number of parallel operator executions. With tens of cores on CPUs, finding the optimal degree of parallelism of a query plan using heuristic and cost model based exchange operator approach is difficult [30]. Some of the prominent problems are a huge multi-core aware plan search space, parallelism aware accurate cost model estimations, and the optimal placement of exchange operators in the plan. The degree of parallelism problem becomes even more difficult under a concurrent workload due to competition for shared resources, such as CPU cores, memory, and memory controllers. This forces many systems to take a conservative approach towards plan parallelization decisions, as a sub-optimal parallel plan could often degrade performance. Often a serial plan is preferred as long as it ensures a robust performance [19].

For example, consider Figure 5.1, which shows execution of three TPC-H heuristically parallelized queries for different DOP under a heavy concurrent CPU bound workload, which ensures 0% CPU core idleness (Scale factor 10 on 256 GB RAM with 32 hyper-threaded cores). The queries show varying performance under different DOP. The traditional plan generation approaches based on heuristic and cost model [61] fall short, as the plans do not reflect run-time resource variations, making them sub-optimal under a concurrent workload.

We introduce *adaptive parallelization*, a new mechanism to generate *range par-*

titioned parallel plans using query execution feedback, while taking into account the run-time resource contention. Adaptive parallelization generates a better plan (P1) from an old plan (P0) in a greedy manner, by parallelizing the most expensive operator from P0, under repeated query invocations. The inspiration is derived from the observation that in real world systems the same query templates get reused multiple times only changing some parameters. Starting with a serial plan, each successive query invocation results in a new parallel plan, until a near minimal execution time parallel plan is detected, which ensures a near optimal DOP. Adaptive parallelization under concurrent workload reflects resource contention, making adaptive parallelized plans resource contention aware [12]. The success of adaptive parallelization depends on its ability to converge quickly, while ensuring a near minimal execution parallel plan.

Adaptive parallelization also allows to analyze the relation between DOP and multi-core utilization. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing. Maximum multi-core utilization however need not improve performance, as it might lead to memory bandwidth pressure due to parallel operator executions [107]. Hence, finding the right balance between the DOP and multi-core utilization is important. Since adaptive parallelization generates new parallel plans incrementally, it enables us to analyze the relation between DOP and multi-core utilization. Adaptive parallelized plans have minimal multi-core utilization and a near optimal degree of parallelism, which helps in achieving better response time during concurrent workloads.

5.2 Contributions

Both heuristic based parallelization and cost model based parallelization do not generate an optimal parallel plan due to their inability to identify the correct degree of parallelization. Adaptively parallelized plans overcome this limitation. We summarize our main contributions as follows.

- We introduce *adaptive parallelization*, a new execution feedback based parallel plan generation technique, that ensures a near optimal degree of parallelization. We show that near optimal degree of parallelization is not always equal to the number of CPU cores, however, could vary between one and the total number of CPU cores.

- We introduce an adaptive parallelization convergence algorithm for different scenarios, that finds a good plan in minimal convergence runs.

- We analyze the parameters affecting the speedup of the core relational algebra operators. We focus on the select and the join operator.

- A near optimal degree of parallelization allows adaptive parallelized plans to show up to five times response time improvement compared to heuristically parallelized plans.

5.2.1 Outline

The Chapter is structured as follows. In Section 5.3 we describe the architecture of adaptive parallelization. We also provide parallelization heuristics for operators and illustrate the dynamic partitioning scheme and discuss related problems. Section 5.4 describes the convergence algorithm to find the near minimal execution parallel plan along with various convergence scenarios. In Section 5.5 we provide a detailed experimental evaluation. Related work is described in Section 5.6. We conclude citing the major lessons learned in Section 5.7.

5.3 Architecture

Adaptive parallelization can be used by any columnar database system as long as its plan representation allows identification of individual expensive operators.

5.3.1 Run-time environment

It consists of a scheduler, an interpreter, and a profiler. The scheduler uses a data-flow graph based scheduling policy, where an operator is scheduled for execution once all its input sources are available. While an interpreter per CPU core executes the scheduled operators, the profiler gathers performance data on an executed operator basis. The profiling overhead is minimal due to vectorized nature of execution. The profiled data consists of operator's execution time, memory claims, and thread affiliation id. Cost model based plan generation approaches often suffer from incorrect cardinality estimates. We use a heuristic plan generation approach where parallelization decisions are based on execution time feedback, without a need for operator's cardinality statistics.

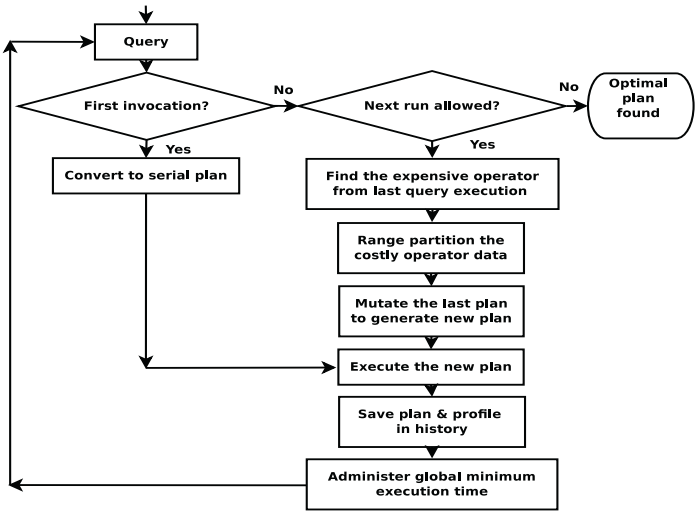

Figure 5.2: **Adaptive parallelization work-flow.**

The execution time is a good metric for parallelization decisions as it reflects the system state such as the memory bandwidth pressure and the processor usage. Though the presence of system noise might affect execution time, such disturbances level out during adaptation.

5.3.2 Infrastructure components

The Adaptive Parallelization **(AP)** infrastructure is implemented using the following components a) operator stubs to morph a plan based on past behavior, b) the plan administration policies to choose a suitable plan from the plan history, and c) the AP convergence algorithm, which we describe in Section 5.4.

5.3.3 Work-flow

The adaptive parallelization work-flow is summarized in Figure 5.2. The first phase is similar to most systems [82] where an optimal serial plan (Figure 5.3 Plan 1) is generated. Our approach differs in the second phase where the the query is cached, plan is fed to the framework, executed, and the profiling information such as the query execution time, the operator execution time, the number of invocations, etc. are stored. On the next query invocation a new parallel plan (Figure 5.3 Plan 2) is derived from the immediate old plan (Plan 1) by parallelizing the most expensive operator (Select on input A). The AP process iterates by invoking the same query again and generating parallel plans in an incremental manner by parallelizing the most expensive operators in successive steps. The number of iterations to find the minimal execution time parallel plan is controlled by a convergence algorithm described in Section 5.4.

5.3.4 Why a feedback based approach?

Like parallel databases, multi-core CPUs make the parallel plan search difficult [101]. The main problem is finding the *optimal number of partitions* per operator for an *optimal input serial plan*. Finding an optimal input serial plan is out of the scope of this paper. During parallelization when an operator's data is partitioned, there are combinatorial possible choices for the partition size. For example, in the worst case, each operator's data can be partitioned in a single tuple, such that the total number of operators equal the number of tuples. In the best case a single operator could work on the entire non-partitioned input. The possible partition size choices for different operators represent multiple parallel plans with different execution times, making this a combinatorial plan search problem. The plan search space exploration is usually done using a combination of both the heuristic and the cost model based approach. It allows to prune the search space for an efficient search. Overall, finding an optimal multi-core aware parallel plan using traditional approaches is difficult. In comparison the feedback based approach we propose is relatively easy, as the assumption is the input serial plan we start with is an *optimal plan.* Since the approach explores the search space in a guided way by parallelizing only the most expensive operator, we avoid a large space of uninterested plans.

5.3.5 Plan mutation

We refer to the process described in the work-flow as *plan mutation*. Plan mutation can be guided by different policies. In this paper we use parallelization of the *expensive operator* in a plan as the guiding principle. An operator is considered *expensive* if its execution time is the highest amongst all operators. Based on the complexity, we categorize mutation in three types as Basic, Medium, and Advanced.

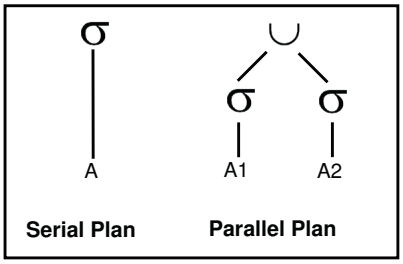

Figure 5.3: **Basic mutation select operator.**

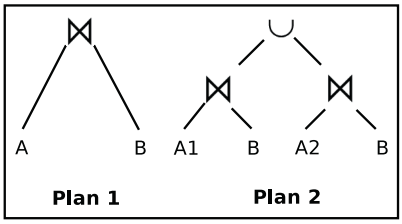

Figure 5.4: **Basic mutation join operator.**

Basic mutation

Basic mutation involves parallelization of an expensive operator by introducing two new operators of the same type, called expensive operator's **cloned operators**. The cloned operators work on the expensive operator's partitioned data. Partitioning is cheap when it involves no data copy, but introducing range partitioned sliced view of the columnar data. (Value / hash based partitioning needs the presence of a partition operator, about which we discuss in Section 5.6) An *exchange union* operator

(either a newly introduced or an existing one) combines the result of the cloned operators. In Figure 5.3 we see one such example for select operator parallelization.

The two most popular algorithms for the join operator are the hash join and the sort merge join. We analyze the hash join implementation as it suits most workloads due to the omnipresence of non-sorted data. We consider adaptive parallelization of the join operator plan (Figure 5.4 Plan 1) when only the larger (outer) input is split into equi-range partitions on consecutive runs. Figure 5.4 Plan 2 shows the parallelized plan with the two new join cloned operators. An exchange union operator combines the output of the cloned operators.

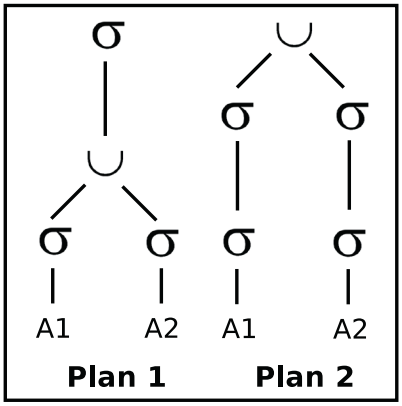

Figure 5.5: **Medium mutation.**

Medium mutation

Medium mutation handles plan parallelization when the *exchange union* operator (U) itself turns out to be expensive, as a result of intermediate data copying due to low selectivity input. This mutation stage arises, when the exchange union operator is introduced as a result of the basic mutation.

Figure 5.5 shows one such example where Plan 1 with an expensive exchange union operator is mutated into Plan 2. The mutation process involves propagating the inputs to the exchange union operator, to its data flow dependent operators. The data flow dependent operators are cloned to match the exchange union operator's input. Finally a newly introduced exchange union operator combines the result of the cloned operator's output.

Advanced mutation

Advanced mutation involves parallelization of operators such as group-by and sort, that do not exhibit the filtering property (selectivity = 0).

Figure 5.6 shows parallelization of a group-by operator with the advanced mutation. The expensive operator (group-by) is parallelized by introducing two cloned operators on its equi-range partitioned data. Next the aggregation operators such

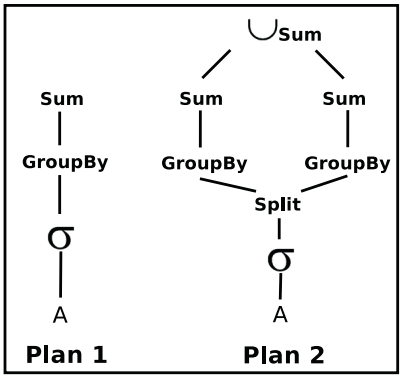

Figure 5.6: **Advanced mutation.**

as sum and average are parallelized by introducing two aggregation cloned operators. The cloned operators (group-by) result is propagated to the aggregation cloned operators (sum). Finally, an exchange union operator combines the parallelized aggregation operators result. Since the aggregation cloned operators always show very high filtering property, the exchange union operator combining their result is cheap.

Summary: A relational operator gets parallelized in two cases. In the first case, the operator itself might be expensive and gets parallelized using either the basic or the advanced mutation. In the second case, operator parallelization occurs as a result of using the medium mutation, where the operator is in the data flow dependent path of the expensive exchange union operator. In both cases identifying and resolving the parallelizable operator's output propagation dependency across the entire plan is an essential step.

The three mutation schemes we described cover all possible mutations as an operator could either get parallelized due to its own expensiveness or as a result of its presence in the data flow path of another parallelizable operator.

5.3.6 Making plans simpler to mutate

Most columnar systems [19, 34, 42, 99] use a simple representation of plan with operators represented using physical algebra. The operators use standardized interfaces for individual columnar data and related argument passing. Column store specific functionality such as operations on multiple columns and tuple reconstruction are mostly hidden away as the internal logic in the execution engine framework. Some column stores like the open-source system MonetDB, however use an abstract language to represent plans [40], where column store specific functionality such as the tuple reconstruction and other columnar operations is exposed in the plan representation itself. Use of operators with different semantics and specialized operators such as the tuple reconstruction operators is common. Figure 5.7 shows one such plan with complex data flow dependencies.

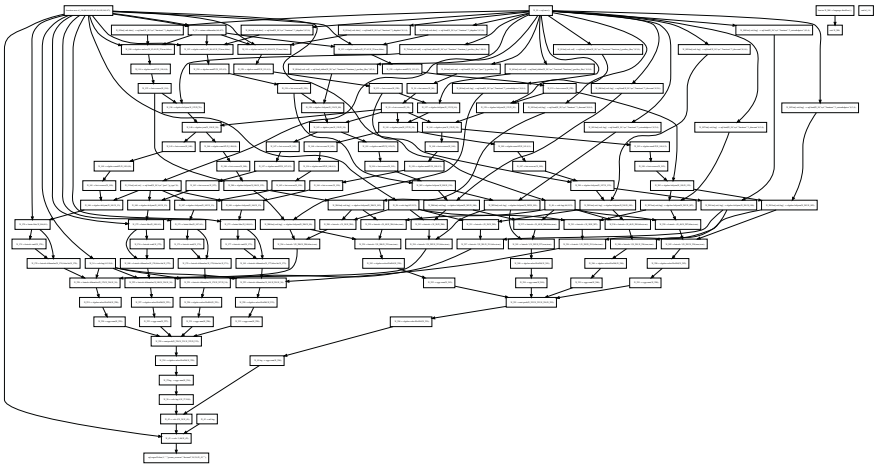

Figure 5.7: **Complex operator dependencies in TPC-H Q14 parallel plan. Rectangles represent operators, and edges between them represent the dependencies. The graph gives a high level perspective of the plan's complexity, abstracting individual operator details. [63] shows graphs where operators are visible.**

Plan mutations using either the medium or advanced mutation involves resolving parallelized operator's propagation dependencies. Hence, care has to be taken to resolve parallelized operator's propagation dependencies. To make plan mutations simpler, modification of some of the operator's semantic representation is needed. We describe the related aspects in the rest of the section.

Adaptive parallelization and operator semantics

Operators can have different semantics depending on primitives being used. Adaptive parallelization could further add more information such as as the partition under use, total number of partitions, etc. Plan mutations thus generate combination of different operator semantics.

For example, consider the case of the filter operator. A simple optimization involves generation of a bit-vector of the filtered tuples, to be fed to another dependent filter operator. Hence, the filter operator can be represented using two primitives depending on the number and the type of inputs. Depending on the data flow dependency, a suitable filter operator gets parallelized during plan mutations.

Adaptive parallelization uses different parallelization rules catered to different operator semantics. Since any operator can be expensive, resulting in its parallelization, the challenge for different mutation schemes lies in how well they are able to resolve the data flow dependencies across different operator semantics.

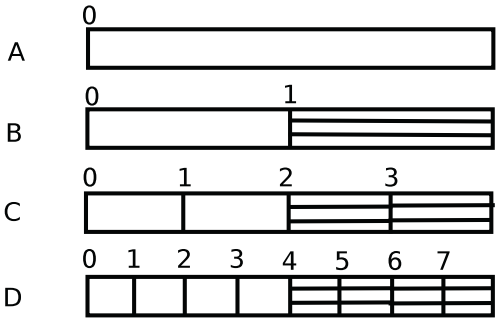

Figure 5.8: **Dynamic partitioning of a column.**

Plan rewriting

One of the techniques to ease the mutation process is to modify the original input serial plan from the SQL compiler using a query re-writer. The re-writer substitutes original operators (for example, aggregation operators and tuple reconstruction operators) with new adaptively parallelization aware operators. These new operators use modified implementation of operators such as group-by, aggregation operators (sum, avg), and sort, by keeping their original semantics, but with changed arguments ordering, to resolve possible operator propagation dependencies. For systems with simple plan representations the operator propagation dependencies due to multiple columns can be handled in the execution engine framework logic.

5.3.7 Adaptive parallelization aware partitioning

In a column store the operators operate on an array or vector representation of the data. For readability, we consider the array representation with range partitioning. It involves creating read only slices on the base or the intermediate column. Creating slices involves marking the boundary ranges for the base or intermediate columnar data and is cheap, as there is no data copying involved. This technique can be also used during vectorized execution where the vectors are derived from the partitioned range of the base and intermediate input. We briefly describe a value based partitioning approach use case in Section 5.6.

Dynamic partitioning

Adaptive parallelization generates dynamically sized partitions on the base or intermediate column, as any operator can be parallelized during successive iterations. In contrast a heuristically parallelized plan often uses a fixed number of partitions based on the available CPU cores. To explain dynamic partitioning of a column using a select operator, we use Figure 5.8.

When the select operator on the column in 5.8A turns expensive, the column is sliced in two partitions represented by 5.8B. When the select operator on partition 1 in 5.8B turns expensive, two new partitions are introduced, represented by 2nd and 3rd in 5.8C. Now there are three select operators, one on 0th partition of 5.8B and

two on 2nd and 3rd partition of 5.8C. When the select operator on 2nd partition in 5.8C becomes expensive, it is divided further and two new partitions 4th and 5th in 5.8D are introduced. So now there are total 4 select operators working on 0th partition of 5.8B, 3rd partition of 5.8C and 4th,5th partition of 5.8D. Please note that the partitions are of different sizes and their boundaries are aligned on the base column in 5.8A. Maintaining the alignment during dynamic partitioning is important, as misalignment could lead to problems such as a) repetition of data b) omission of the data across different operator partitions. Thus, dynamic partitioning allows the operators to work on different sized partitions of the same column in parallel.

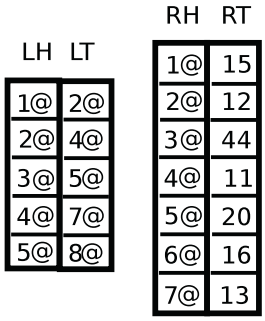

Figure 5.9: **Tuple reconstruction between two columns.**

Dynamic partitioning and tuple reconstruction

Tuple reconstruction is a well known problem in column stores [84] and is implemented as join look-up. Column stores use either early or late materialization strategies for column projection using tuple reconstruction, which involves using row-ids to fetch values from the column that needs projection. For example, consider the columnar representations in Figure 5.9, which shows head (H) and tail (T) columns grouped as Left (L) and Right (R). The head column (LH / RH) contains row-ids, whereas the tail column contains either row-ids (LT) or actual values (RT). The @ indicates a row-id. When the head column (LH / RH) contains consecutive row-ids, it is not materialized and used as a virtual column. During tuple reconstruction, the row-ids in the left tail (LT) are used as an index in (RH) to fetch the corresponding values from the right tail (RT). For example, row-ids 2, 4, 5, 7 from LT are probed in the RH, whose corresponding values in RT are 12, 11, 20, and 13.

One important aspect is the effect dynamic partitioning has on the tuple reconstruction due to possible misalignment between LT and RH. When row-ids from LT are used as an index to fetch values from RT, the row-ids in LT should be a subset of row-ids in RH. If not, then a look-up using row-id in LT, for the row-id index in RH does not exist, resulting in an invalid access.

Since adaptive parallelization generates variable sized partitions, it gives rise to different alignment scenarios as shown in Figure 5.10 (B,C,...F). Consider the misalignment example in Figure 5.9. Here LT start row-id=2, which is greater than

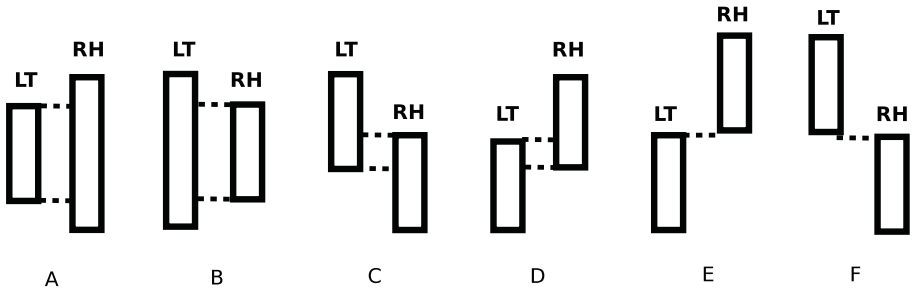

Figure 5.10: **Different alignment scenarios during tuple reconstruction due to dynamic partitioning.**

RH start row-id=1, and LT end row-id=8, which is greater than RH end row-id=7. Hence, LT's upper boundary starts after RH's upper boundary, whereas LT's lower boundary extends beyond RH's lower boundary as represented in Figure 5.10D. In Figure 5.10 the lengths of columns provide just a logical representation of over and undershooting of boundaries, and do not represent the actual content. To maintain the alignment the lower boundary of LT is adjusted by removing row-id=8, to match the lower boundary of RH. The correct boundary alignment is represented by dashed lines in Figure 5.10D. Adaptive parallelization depending on the operator semantics uses one of the alignment scenarios, to make sure that the partitions align correctly. Fixed size partitions always lead to correct alignment (See Figure 5.10A), resulting in a valid access.

Another important aspect arises when the output of operators working on the dynamically partitioned data is packed together. Here the exchange union operator must maintain the correct ordering to avoid the incorrect results. The correct ordering is maintained, as the operators whose results are packed follow the mutation sequence order, hence the results being packed together follow the same order. Adaptive parallelized plans can become very large due to successive partitioning and operator propagation, which could make partition misalignment related problems, if any, hard to identify and resolve.

Plan explosion

As adaptive parallelization involves propagating the parallelized operator's output on its dependent operators, the plans could quickly grow large. Plan explosion results as a side effect of the exchange union operator removal during the *medium* mutation. For example, when a descendant of the same type of operator stays expensive during successive invocations, it gets parallelized and a single exchange union operator combines the output of all such parallelized operators. As a result the number of input parameters to the exchange union operator can become very large. Eventually if the exchange union operator itself turns expensive, it is removed using the *medium* mutation. This leads to a plan explosion, as the medium mutation propagates inputs of the exchange union operator on its data flow dependent operators. For each operator in the data flow path, new instructions (operators)

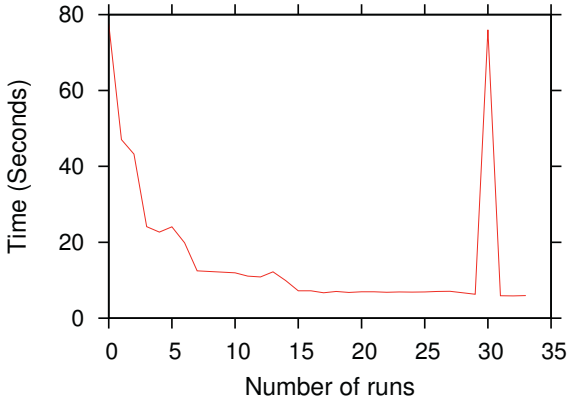

Figure 5.11: **Adaptive parallelization convergence algorithm scenarios for join operator parallelization.**

which equals the number of the exchange union operator inputs are added in the plan. Hence, if the number of input parameters to the exchange union operator is large, the plans could grow very large.

The growth of large plans is suppressed by not removing the exchange union operator if its input parameters cross a certain threshold. The threshold in the current implementation is 15 parameters, chosen on the basis of empirical observations from different parallelization cases. Suppressing the exchange union operator removal however stops further plan parallelization, as the exchange union operator stays the most expensive operator in all further query invocations.

We have described adaptive parallelization aware infrastructure changes so far. Obtaining a minimal execution parallel plan however depends on how fast the adaptive parallelization process converges. In the next section we describe a new algorithm that ensures convergence in different scenarios.

5.4 Algorithm

In this section we introduce the heuristics for the global minimum execution identification from the set of available plans and the corresponding convergence algorithm. The algorithm is loosely inspired by the hill climbing approach [129]. Figure 5.11 shows different cases of the presence of minima, plateaus, and up-hills in the execution times, during adaptive parallelized runs of a join operator plan. We refer to the minimal execution time amongst them as the *global minimum execution (GME)*. Like most systems that generate parallel plans, our base assumption is an optimal input serial plan. Hence, the focus of parallelization is to identify the optimal number of partitions for operator's data in the input plan. Problems such as sub-optimal parallel plans due to poor join ordering which might require backtracking are not considered, as the input is an optimal serial plan.

The convergence algorithm should be able to find the GME in all cases of minima, plateaus, and up-hills in the execution time, and converge in minimal number

of runs. Next we formally define the GME first, the convergence algorithm next, and then illustrate different convergence scenarios.

5.4.1 Global minimum execution (GME)

As the runs progress, the GME is the minimal execution time amongst so far observed runs, and keeps on changing, during an active adaptive parallelization instance.

We denote the current run's execution time as $CurExec$. The *execution time improvement (CurExecImprv)* at the current run is calculated with respect to 0th run's (SerialExec) execution time.

$CurExecImprv = |(SerialExec - CurExec)|/SerialExec.$

To calculate the first GME improvement, we initialize GME to the first run's execution time after the serial execution (0th run).

$GMEimprv = |(SerialExec - GME)|/SerialExec.$

As the runs progress, new GME needs to be identified. A new run's execution time becomes the new GME, if the run's execution time improvement is better than the current GME's execution time improvement by a certain threshold.

$GME = CurExec$
$\{if(CurExecImprv - GMEimprv) > threshold\}.$

As the runs progress, the new execution times can be slightly lower than the existing GME, hence, selecting the correct threshold is important in discarding such new execution times, which otherwise can become the new GME. For example, consider a hypothetical adaptive parallelization instance. Let CurExecImprv at the 8th run be 96%, GMEimprv at the 3rd run = 90%, and threshold = 5%, then *(CurExecImprv - GMEimprv) >5*. Hence, the Current run Execution time at the 8th run is considered the new Global Minimum Execution (GME). Correct tuning of the *threshold* parameter is thus crucial as it helps to discard multiple possible GMEs and to chose the optimal execution time amongst them, as the new GME.

Finding GME can be also difficult due to the presence of many local minima, about which we illustrate next.

The global minimum detection problem:

The problem can be formally stated as finding the global minimum execution from many local minima that occur as the runs progress. When the execution time of a run is more than its previous run, a local minimum results at the previous run. For example, a local minimum occurs at the 4th run in Figure 5.11. The convergence algorithm has to overcome many such local minima during its exploration of the global minimum. We use the rate of improvement in the execution time of the runs as a heuristic, to avoid the local minimas.

The execution time of consecutive runs could improve or worsen depending on the run-time conditions (execution skew, operating system interference), giving rise to positive or negative *rate of execution time improvement* **(ROI)**. The ROI of a run

is defined with respect to its previous run's execution time *(PrevExec)*. We define ROI as follows.

$ROI = (PrevExec - CurExec)/MAX(CurExec, PrevExec)$.

In Section 5.4.2 while describing the core convergence algorithm, we illustrate how to use ROI to avoid local minimas. Finding GME is difficult, however, another equally difficulty task is to find it in the minimal convergence runs, about which we illustrate next.

The minimal run convergence problem:

The problem can be formally stated as finding GME in a minimal number of runs, during consecutive query invocations. Too few runs have a risk of non-occurrence of the global minimum and the algorithm converging on a local minimum. Too many runs might ensure a global minimum at the cost of a slow convergence. Hence, finding the right balance between the minimum convergence runs and the GME is of prime importance for the convergence algorithm.

5.4.2 Convergence algorithm

We describe the convergence algorithm using the context described so far in Section 3. The aim is to find the GME in minimal number of convergence runs. We model the number of convergence runs *(Convergence_Runs)* using the parameters ***credit*** and ***debit***. A credit reflects the number of runs accumulated at each run due to a positive ROI. A debit reflects the number of runs accumulated at each run due to a negative ROI.

$Credit = Credit + (ROI * Number\_Of\_Cores)$.
$Debit = Debit + (|(ROI)| * Number\_Of\_Cores)$.

The value of *(credit - debit)* at each run reflects the balance (*Convergence_Runs*) available for the system to converge. Hence, the next run is allowed only if the balance is positive i.e. *((credit - debit) >0)*.

$Convergence\_Runs = Credit - Debit = f(ROI)$.

The algorithm starts with the value of credit = 1 and debit = 0. When parallelism reduces the execution time, the ROI of the first run is positive and very high (Figure 5.11 - The algorithm starts with the 0th run). With an increase in runs, the ROI decreases. During the initial few runs the algorithm should ensure availability of sufficient runs as a balance, to avoid premature convergence. During the later runs, as the ROI slows down, the algorithm should ensure as few balance runs as possible, to ensure fast convergence. From the formula above, as both credit and debit are dependent on ROI, they are a function of ROI, which makes the Convergence_Runs also a function of ROI. The algorithm convergence is hence guaranteed, since the heuristic ***Credit - Debit >0***, which decides the available *Convergence_Runs* becomes invalid eventually. Next we describe various convergence scenarios and how the heuristic *Credit - Debit >0* becomes invalid, which guarantees the algorithm's convergence.

5.4.3 Convergence scenarios

We identify three scenarios during which the algorithm should ensure the convergence, 1. No premature convergence in a local minimum before identifying a global minimum. 2. No extended convergence, and 3. The convergence in a noisy environment. We expect these scenarios to cover the entire spectrum as the aim is to find the global minimum, and the possible problems for the convergence algorithms can be its early termination, late termination, and termination during noisy environment. We describe these scenarios next.

No premature convergence

When parallelism improves the execution time, the first run always has a very high ROI (Figure 5.11 - The algorithm starts with the 0th run). Hence, the credit accumulated after the first run is very high with an upper limit of *(Number_Of_Cores + 1)*. This ensures that there are sufficient runs available as a balance in the system during the initial stages to overcome plateaus and up-hills. Each run after the first run contributes more credit, ensuring more runs. This is also analogous to the concept of accumulation of the *potential energy* by a body when it falls from great heights. The greater the height, the higher the potential energy. The energy allows the body to keep moving in plateaus and climb high hills, as long as there is a balance energy.

No extended convergence

Accumulation of high credit in the few initial runs on a stable system could result in a state where the algorithm *never* converges. In a stable system the execution time variations are minimal, leading to fewer *debits* being made. In such a system, the proportion of accumulated *credit* will always be much higher than the accumulated *debit* after a few initial runs. For example, consider Figure 5.11. After 15 runs the ROI is minimal, ensuring that no new significant credit or debit is introduced. However, the accumulated credit till 7 runs is very high, as the ROI till 7 runs is very high. This situation leads to non-convergence as there are always balance runs available i.e. *(credit - debit<= 0)* is never true.

Leaking debit: To ensure the algorithm converges in a finite number of runs we introduce the concept of *leaking debit*. In this scheme after a *threshold* on the number of runs is crossed, a constant debit gets deducted from the available credit at each run. *It ensures the available credit is drained to 0, so that the algorithm converges in a finite number of runs.* Hence, *leaking debit* is a function of the available *credit* at the threshold run. The threshold run value is calibrated to be the *Number_Of_Cores* on the CPU. It ensures at least those many runs are used to find the optimal execution time. The *Leaking_Debit* is calculated by dividing the available credit at the threshold run amongst the possible remaining number of runs during the global minimum search.

$$Remaining\_Runs = Extra\_Runs * Number\_Of\_Cores.$$
$$Leaking\_Debit = Credit / Remaining\_Runs.$$

Based on plan complexity, some queries converge early, while some take longer after crossing the *threshold run* reference. To avoid premature convergence, the system specific tunable parameter *Extra_Runs* is used, which ensures that the *remaining number of runs* to search the global minimum are sufficient. Note that *Remaining_Runs* is just an approximate bound. Plan representations vary considerably across systems. Hence, based on empirical observations from different parallelization cases, and multiple experimental runs (five), for the current platform, Extra_Runs=eight is considered a safe boundary value to avoid the premature convergence. Higher values result in an extended convergence.

Convergence in a noisy environment

Depending on the stability of the run-time environment (operating system process interference, memory flushes, etc.) the execution time of an individual run could vary considerably. The execution time of some of the runs in a noisy environment is often greater than the serial plan execution time. One such peak is visible in Figure 5.11 at the 30th run. Most peak executions are followed and preceded by a normal execution. If care is not taken such peaks will make the algorithm halt *immediately* as the debit due to peak ascent will be higher than the accumulated credit. Hence, the algorithm should converge gracefully in such a noisy environment.

Our solution is to mark all such unique peaks as outliers, and ignore their presence. The algorithm incorporates this by allowing the immediate next run to execute. This ensures the balance runs stay unaffected, as the *debit* made during the peak *ascent* is compensated by an equivalent *credit* during the peak *descent*, during the next run. Concurrent workload could also affect the convergence, however, tuning the *Extra_Runs* parameter to find the leaking debit should take care of it.

Global minimum plan identification proof

The convergence algorithm should ensure a global minimum plan while converging in a reasonable number of runs. The lower bound on the convergence runs is *Number_Of_Cores + 1*, while the upper bound approximates between *(Number_Of_Cores + 1 + Remaining_Runs)* and extra runs added to the previous upper bound, if any, due to a large credit accumulation. The convergence runs are directly influenced by the *Leaking_Debit*, and credit / debit accumulation.

The global minimum plan's existence beyond the upper bound of the convergence runs is not possible. We provide a proof by contradiction. If such a plan exists then its execution should be significantly better. In that case the corresponding expensive operator should have been identified much earlier, even before the first upper bound on the convergence runs is reached. If multiple such plans exists, then that indicates improved execution with each run. Such improvement should then add extra runs (more credit) to the first upper bound on the convergence runs, which would prolong the global minimum search further, to find a more optimal plan. Hence, no matter the situation, a near global minimum plan is identified in the available convergence runs. In all the convergence scenarios when the heuristic ***Credit - Debit >0***, that decides available *Convergence_Runs*, turns invalid, the algorithm converges.

Table 5.1: **System configuration**

CPU	Sockets / Threads	L1 / L2	Shared L3	Mem	OS
Xeon E5-2650@ 2.0GHz	2 / 32	32 / 256KB	20MB	256GB	Fedora 20
Xeon E5-4657Lv2@ 2.4GHz	4 / 96	32 /256KB	30MB	1TB	Fedora 20

5.5 Experiments

Adaptive parallelization is implemented in MonetDB, being the only full fledged open-source columnar system, with memory mapped columnar representation for the base and the intermediate data. The operators are represented in an intermediate language called MonetDB Assembly Language (MAL) [40], with their implementation in C. The operators have variable number of arguments depending on their semantics, and form complex data flow patterns in MAL plans, as shown in Figure 5.7.

Table 5.1 summarizes our experimental hardware platform, which consists of two types of machines, with two and four socket CPUs each. All experiments, unless mentioned, use in-memory data (without disk IO) on the two socket machine. Heuristic parallelization unless mentioned uses 32 threads. Each graph plots an average of four runs of the same experiment. We use the four socket machine to test one of the workload's scalability from NUMA perspective.

The experimental section is divided into two broad categories. In the first we analyze how parallelization gets affected by various operator level parameters. In the second we analyze it at the SQL query level. We use a mix of micro-benchmarks, simple, and complex SQL queries to gain parallelization behavior insights.

We use TPC-H and TPC-DS workload for SQL query level performance comparison. We observe the TPC-H isolated execution of both the adaptive and the heuristic parallelization shows similar performance. However, adaptive plans are better as they use fewer number of cores, which helps during concurrent workload. Adaptive plans show better performance than the heuristic plans for the TPC-DS workload isolated execution, due to optimal number of partitions, and the presence of the skewed data. In the rest of the section we describe the experimental details.

5.5.1 Operator level analysis

Since adaptive parallelization uses expensive operator parallelization as a heuristic, analysis of an operator's behavior gives insights into parallelization issues such as the execution skew. The execution skew occurs when at least one of the parallelized operators takes longer to execute than the rest. An operator's execution time varies on the basis of type of computation, amount of data being read / written, type of data access (serial / random), and memory hierarchy of the access (cache / main memory / disk IO). We analyze some of the factors that influence these parameters next.

Data skew

This experiment highlights the role of *dynamic* sized partitions to avoid execution skew during parallelized execution, when the data distribution is *non-uniform*

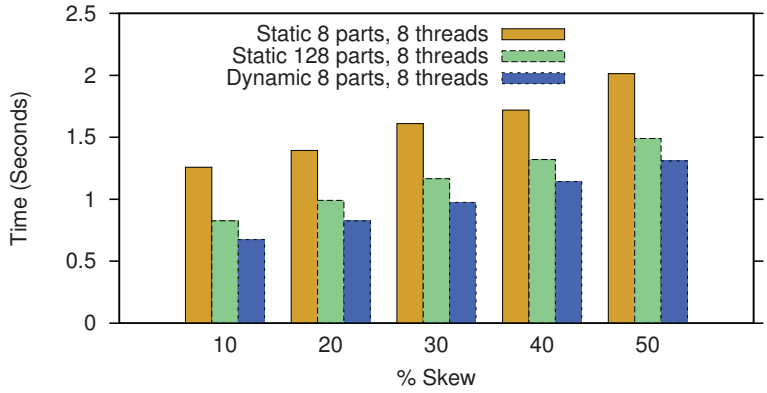

Figure 5.12: **Parallelized select operator execution on skewed data using static and dynamic (adaptive) sized partitioning. The second bar indicates a work stealing based approach.**

(skewed). The execution skew occurs when at least one of the parallelized operators takes longer to execute than the rest.

Static partitioning (equi-range partitioning) of skewed data leads to execution skew as some partitions have more matches than the rest. Adaptive parallelization performs well in skewed data scenario as the operator with the skewed partition turns expensive, and gets parallelized until expensiveness balances out.

Figure 5.12 shows the execution time when parallel select operators work on statically or dynamically (adaptively) partitioned skewed column of type *long (8 bytes)*. The number of tuples in the input column are 1000 million **(M)** (size = 8GB). Figure 5.13 shows the column's data distribution with 500 million random tuples in the first half. The second half contains skewed data with 5 sequential clusters of 100 million identical tuples. We vary the select operator's condition to generate the execution skew.

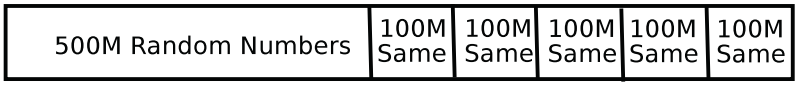

Figure 5.13: **Data distribution for a skewed column.**

Figure 5.12 shows execution with 8 threads on 8 dynamically sized partitions (black) is up to 60% better than the execution with 8 threads on 8 static partitions (khaki). One may argue that the work stealing approach [38] could solve the problem of execution skew due to the static partitions. We analyze it by creating a large number of smaller partitions (128) operated upon by 8 threads. Large number of smaller partitions allows those threads that finish work early to operate on remaining partitions, while threads on skewed partitions stay busy. Identifying the optimal combination of static partitions and threads is however non trivial, as in some cases more partitions might lead to plan blow-up resulting in scheduling overheads. In contrast we observe that the dynamic sized (adaptive) partitioning approach with 8 threads and just 8 partitions fares competitively with static 8 threads, 128 partitioned approach.

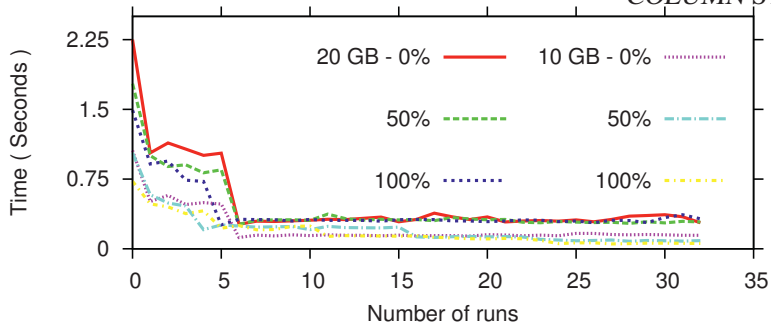

Figure 5.14: **Effect of variations of data size (20GB,10GB) and selectivity on the speed-up of the parallelized *Select* operator plan.**

Summary: Skew handling is a natural property of adaptive parallelization. It is a result of dynamically sized range partition creation, and a side effect of the expensive operator parallelization heuristic.

Selectivity, Input size and Exchange union operation

In this experiment we analyze the effect of selectivity, input size and the exchange union operation on the speed-up of select and join operators. *Speed-up* is defined as the ratio of serial to parallel plan execution time. The experiment also allows analysis of increasing the number of threads from 1 to 32 and its effect on the speed-up. This is possible since with each iteration one more partition gets added and is available for one more execution thread (from a pool of 32 threads) to operate in parallel.

Exchange union operation

Many systems use the *exchange operator* based parallelism [68], where one of the concerns is to identify the correct placement of the exchange operator in a plan, to minimize its overhead [30]. Most systems use a cost model based approach for this decision. A good example is [142], where Vectorwise is shown to have a limited speed-up due to the exchange operator overheads.

As the exchange union operator combines parallelized operator's result, its expensiveness varies depending on the size of the data being packed. Low selectivity reflects more matching data, hence more data to be packed. The packing overhead is minimized by pushing the exchange union operator as high as possible. It ensures the final data to be packed is relatively small, as it gets filtered by the intermediate operators.

Adaptive parallelization enables to analyze the exchange union operator's placement with successive iterations of parallelized plans, as it directly affects the speed-up. In the next two experiments we observe that the AP plan's speed-up is comparable to the speed-up of the heuristically parallelized (HP) plans. The speed-up gets hindered due to operator dependencies that form critical paths, which can not be parallelized. **The real benefit for AP plans is in its optimal multi-core utilization due to less partitions**, which ensures improved *concurrent* execution performance, as we describe in Section 4.2.3

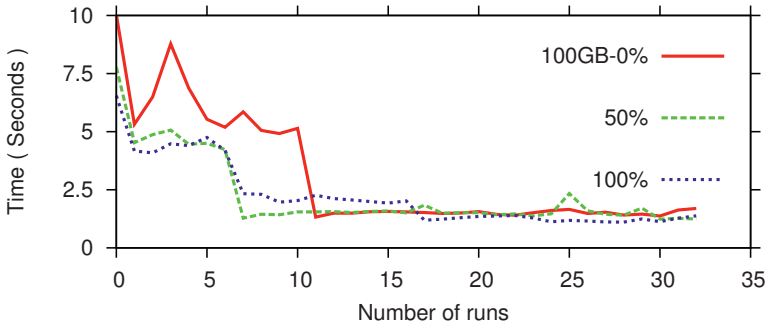

Figure 5.15: **Effect of variations of data size (100GB) and selectivity on the speed-up of the parallelized *Select* operator plan.**

Select operator adaptive parallelization

We use query 6 from the TPC-H benchmark to analyze the speed-up of the select operator (See Table 5.2). Query 6 is a simple query with only selection predicates on the *Lineitem* table. We vary the selectivity by varying the parameter *l_quantity* from the selection predicate. Figures 5.14 and 5.15 plot the execution time of adaptively parallelized plans on the Y axis with respect to iterations (X axis), when selectivity is varied from 0% (all output) to 100% (no output), and scale factor is varied from 10 GB to 100 GB.

From Table 5.2 as the selectivity increases the speed-up decreases. During low selectivity a single select operator in a serial plan writes a large number of output tuples, as compared to its parallel plan counterparts. This results in the large speed-up as serial execution time is much higher, whereas parallel execution time is much lower. During highest selectivity (100%) since there is no output the serial execution is less expensive as compared to 0% selectivity serial execution. This results in lower speed-up. The speed-up increases with a decrease in the input size. This is a result of lower minimum time during parallel execution, due to less input data. With increased selectivity the speed-up for AP is less compared to HP. This is due to the presence of less expensive exchange union operators, which do not get pushed higher in the plan.

Table 5.2: ***Select* operator plan speed-up (compared to serial execution) using adaptive and heuristic parallelization.**

	AP = Adaptive, HP = Heuristic parallelization					
	Selectivity					
	0%		**50%**		**100%**	
Size (GB)	**AP**	**HP**	**AP**	**HP**	**AP**	**HP**
100	10	10	8.5	10	7	9
20	10.5	12	8.5	12	8	12
10	16	11	14.5	11	12	9.5

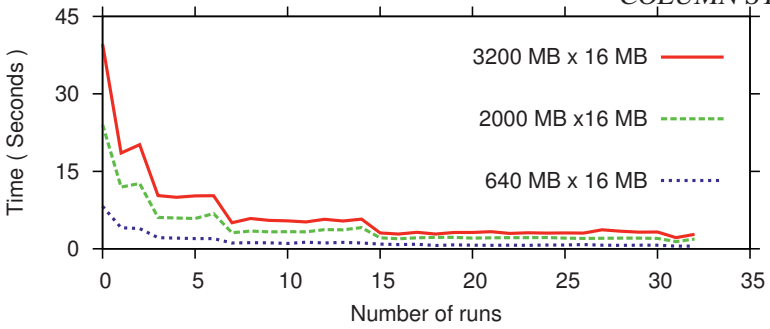

Figure 5.16: **Effect of variations of data size on the speed-up of the parallelized *Join* operator plan. Outer input partitioned and inner input used to build hash table.**

Join operator adaptive parallelization

For the join operator (hash) plan parallelization analysis, we partition the outer input and build up the hash table on the inner input. We use a micro-benchmark for a fine grained control, where the outer input has 400 M, 250 M, and 80 M (M = Millions) random tuples of type *long (8 bytes)*, and the inner input has 8 M and 2 M tuples. The outer inputs stay larger than the inner input of size 16 MB (2 M tuples) even after 32 partitions (threads on CPU). The 16 MB input fits in the shared L3 cache (20 MB).

Figure 5.16 shows the join operator plan speed-up and Table 5.3 quantifies it. The speed-up of 16 MB input join is more than the 64 MB input join, as the 16 MB input join's hash table fits partially in the L3 cache (20 MB), which improves the probe phase, due to reduced cache thrashing. Speed-up also decreases as the outer table size decreases, as the serial execution time is directly proportional to the outer table size. For all sizes the best speed-up is obtained when the number of partitions are 32, with 32 threads (hyper-threading enabled). Maximum speed-up observed is around the number of physical cores (16). Both AP and HP show a similar performance unlike the previous select operator plan analysis case, as the join plan contains only join and pack operators.

Table 5.3: ***Join* operator plan speed-up (compared to serial execution) using adaptive and heuristic parallelization.**

	AP = Adaptive, HP = Heuristic parallelization			
Size (MB)	**64**		**16 (Smaller Input)**	
(Larger Input)	**AP**	**HP**	**AP**	**HP**
3200	15.75	14	18.5	18
2000	15	13.5	17.75	17.75
640	13.75	13	17	15

Summary

Adaptive parallelization works for both the select and the join operator and these operators scale linearly with the number of physical cores. Input size, selectivity,

and properties such as cache consciousness affects the speed-up.

Operating system noise

This experiment highlights the influence of the operating system's interference on an operator's cost. In an ideal and stable environment each operator working on an equi-range partition with an uniform data distribution should exhibit similar execution cost.
However, when the operating system noise (background process scheduling, memory flushes, IO, etc.) interferes with the execution of an individual operator, the execution time increases. This results in the individual operator execution skew, thereby degrading parallelization performance. The execution time at the 5th run in Figure 5.11 is more (uphill) than the 4th run, as a consequence of the operating system noise interference. The 30th run in Figure 5.11 is an outlier and a result of disk IO due to memory flush. Isolating operating system's interference is difficult. Related problems and solutions can be found from the work on multi-tenant database-as-a-service in [115].

Having considered individual operator level analysis so far, in the next section we focus on holistic complex SQL query level analysis from execution performance and convergence perspective, in the context of TPC-H benchmark queries.

5.5.2 SQL query level analysis

Since the TPC-H benchmark is considered the de-facto workload for performance comparison, in this section we use a subset of queries (see Table 5.4) from TPC-H (scale factor 10). TPC-H has uniformly distributed data. The adaptively parallelized group-by operator implementation at present supports single attribute group-by queries. Hence, we modify some queries so that they have a single attribute group-by representation. Since we use the same set of queries to evaluate multiple parallelization approaches, the comparison is fair. We plot an average of four executions using the experimental set-up described towards beginning of Section 4.

Table 5.4: **TPC-H queries.**

Simple	Q6	Q14			
Complex	Q4	Q8	Q9	Q19	Q22

We compare adaptive parallelization **(AP)** with heuristic parallelization **(HP)**, the default parallelization technique in MonetDB, under isolated execution setting. HP uses parameters such as the number of threads, physical memory size, and the largest table size to identify the number of partitions for the largest table in the serial plan. A plan re-writer generates a parallel plan from a serial plan by propagating the partitions to data flow dependent operators. Though both HP and AP start with the same serial plan, the final parallel plan is different for both techniques as in AP only the most expensive operator gets parallelized unlike in HP, where all possible parallelizable operators are parallelized. In Figure 5.17 the first two bars show HP vs AP performance when queries execute in isolation. All AP queries except Q9 and Q19 show similar performance as HP. Q9 and Q19 show a degraded performance due to the presence of some non-parallelizable operators, which prolong the

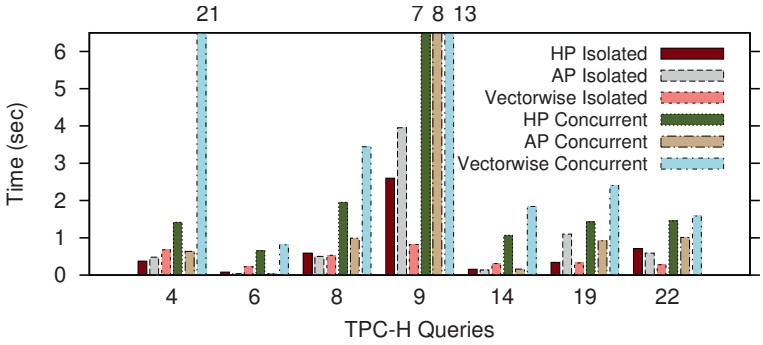

Figure 5.17: **Heuristic vs adaptive parallelization performance in isolated and concurrent workload environment for MonetDB and Vectorwise.**

query execution. Though the execution performance of adaptive plans is similar to the heuristic plans, the adaptive plans are better as they use much fewer number of partitions (See Table 5.5). It helps during concurrent workload execution, where adaptive plans exhibit better execution performance due to better resource utilization.

Table 5.5: **AP and HP Q14 plan statistics.**

	AP	HP
# Select operators	10	65
# Join operators	16	32
% Multi-core Utilization	35	75

TPC-DS queries

TPC-DS benchmark has 25 tables, out of which 6 tables are relatively large (above 1GB in size), in a scale factor 100 dataset. The benchmark supports 99 query templates. We use a few modified queries. These queries are a subset of the original TPC-DS queries and are chosen such that they contain the large tables and a few smaller dimension tables. Since we compare both the adaptive and the heuristic parallelization technique with the same queries, the comparison gives a perspective of their respective performance.

We experiment on both the two socket and the four socket machine (See Table 5.1 for configuration) with 100GB dataset, to get a perspective of the NUMA effects. Graphs in Figures 5.18a and 5.18b show the comparison. Adaptive plans exhibit a maximum of 5 times better performance compared to heuristic plans, which can be attributed to *correct partitioning by adaptive parallelization compared to heuristic parallelization and the skewed data distribution.* The execution time for both two and four socket machine shows similar time, which indicates minimal NUMA effects. As authors in [65] observe, since MonetDB uses a memory mapped representation for the buffer data, as the number of partitions increase, we expect them to get assigned to the memory modules of the sockets on which operator execution gets scheduled. We also observe a limit on the execution improvement, even

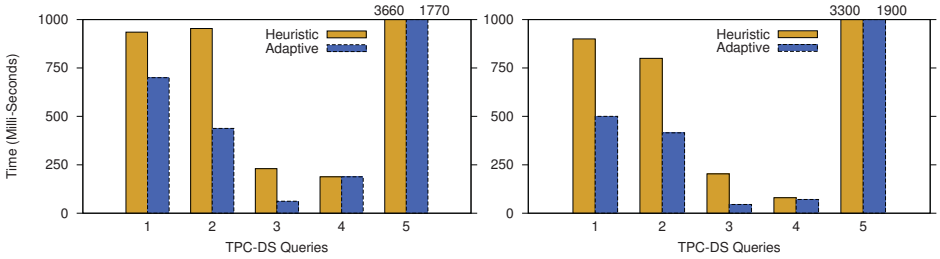

Figure 5.18: **Isolated execution performance of TPS-DS queries on a) 2 socket machine with 2.00 GHz CPU b) 4 socket machine with 2.40 GHz CPU, on 100GB data.**

though a higher number of cores are used, which indicates increased parallelism need not improve performance beyond a threshold.

Concurrent workload execution

This experiment highlights the effectiveness of adaptively parallelized plans compared to heuristically parallelized plans in a concurrent workload setting. Concurrent query executions in batch workload leads to resource contention, which in turn affects the degree of parallelism of individual queries under execution. Resource contention varies with random workload, however for the current set-up we consider a homogeneous concurrent workload. In Figure 5.19 the 4th and 5th bar shows HP vs AP execution under a concurrent workload. The workload consists of random simple and complex queries from the TPC-H benchmark, where 32 clients fire queries repeatedly. AP Q8 shows 50% improved execution compared to HP Q8. Simple queries such as Q6 and Q14 show around 90% execution improvement in AP. HP plans have too many partitions compared to the AP plans as shown in Table 5.5 AP plans also reflect the resource contention through execution feedback. Hence, AP plans are more robust and better performing under a concurrent workload, compared to statically generated HP plans. In [12] we discuss HP vs AP plans comparison under different concurrent workload resource contention scenarios in a detailed manner.

Comparison with Vectorwise

We compare the concurrent workload performance of Vectorwise (version 3.5.1 with histogram build feature enabled to generate optimized plans), a leading analytical columnar database using vectorized execution [42], with adaptive parallelization in MonetDB. Vectorwise uses cost model based exchange operator dependent parallel plans. The resources are allocated based on the number of connected clients and the system load. During a heavy concurrent workload (32 clients invoking random TPC-H queries repeatedly), the first client's query gets all the resources, while the queries from the remaining clients get less resources based on an admission control scheme. Figure 5.17 shows MonetDB adaptive parallelized

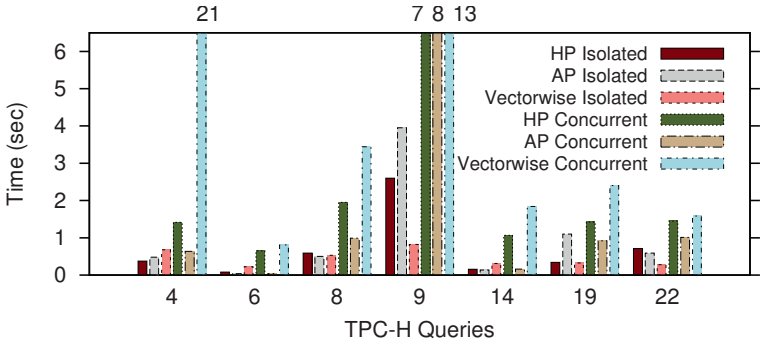

Figure 5.19: **Heuristic vs adaptive parallelization performance in isolated and concurrent workload environment for MonetDB and Vectorwise.**

query execution performance is better than the Vectorwise execution performance, during the concurrent workload. MonetDB does not have explicit resource control based plan generation scheme, which helps in the current case. We hypothesize that as workload queries are invoked repeatedly, Vectorwise queries under analysis execute serially due to lack of resources.

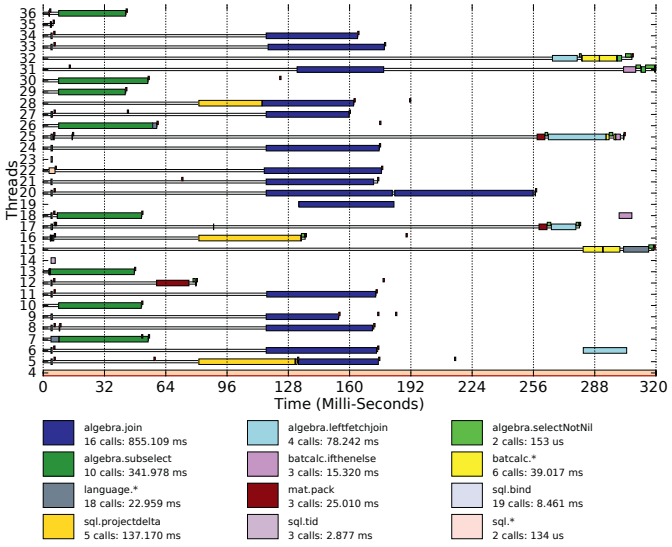

Figure 5.20: **Adaptive parallelization multi-core utilization (35%) during isolated execution of TPC-H Q14.**

Multi-core utilization

This experiment highlights that an AP plan is better than a HP plan from the multi-core utilization perspective. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing. AP ensures

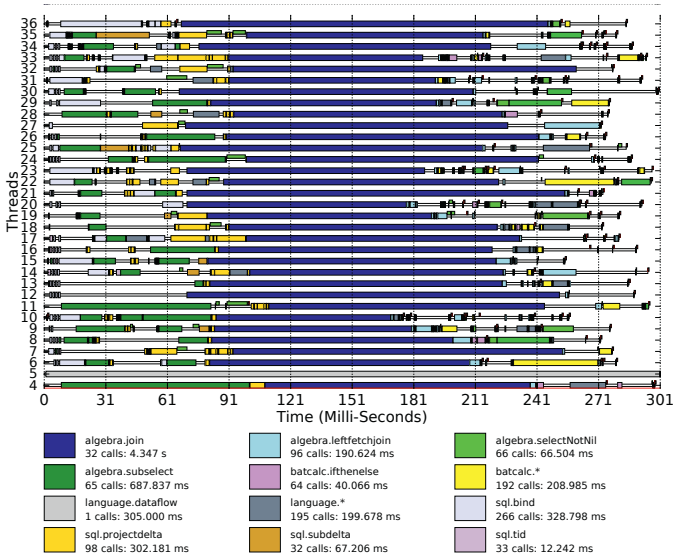

Figure 5.21: **Heuristic parallelization multi-core utilization (75%) during isolated execution of TPC-H Q14.**

minimal multi-core utilization as each operator is parallelized with a different degree of parallelism unlike HP. Figures 5.20 and 5.21 visualize a portion of AP vs HP plan execution of TPC-H Q14, in an isolated execution setting. Similar graphs are described in detail in [64] for further reference. The length of a colored box represents an operator's execution interval (In an operator-at-a-time execution model an operator executes completely.) The black color represents a *join*, the black color represents a *select*, while the brown color represents the exchange union operator. A white space indicates no execution. The amount of white space in Figure 5.20 is much more than in Figure 5.21, indicating lower multi-core utilization for AP. In the HP execution (Figure 5.21) the length of the join operators is much longer than the corresponding operators in the AP execution (Figure 5.20), which hints at the memory pressure.

The degree of parallelism per operator thus influences the overall multi-core utilization. For example, while only ten select operators execute in AP, many more execute in HP. Since AP shows lower multi-core utilization (35%) during isolated execution, the spare resources ensure better response time and throughput during concurrent workload, as we further elaborate in [12].

5.5.3 Convergence algorithm robustness

Adaptive parallelization not only should converge in minimal number of runs, but also should exhibit robustness. The robustness implies during *multiple* adaptive parallelization invocations of a query a) the total number of convergence runs b) the run at which the global minimum occurs, and c) the global minimum execution

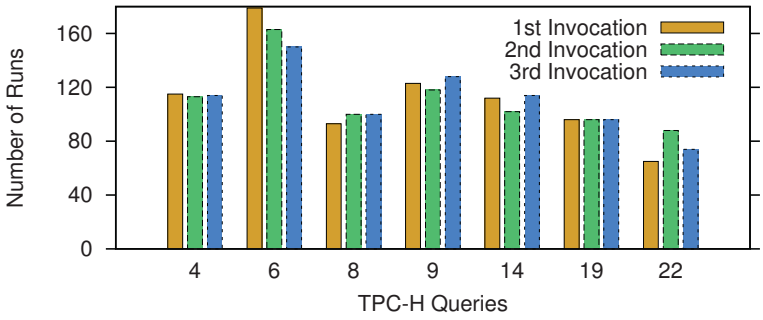

Figure 5.22: **Convergence runs for adaptively parallelized query execution.**

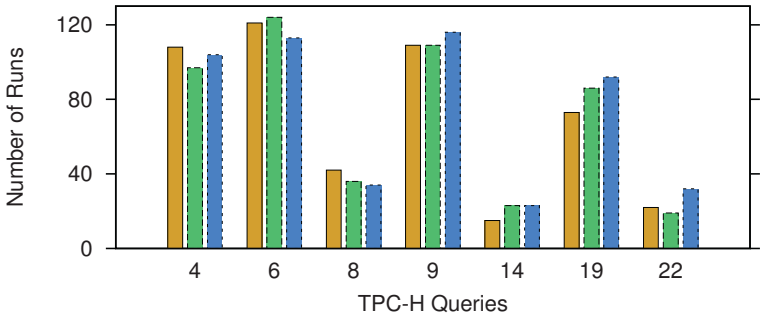

Figure 5.23: **Global minimum run for adaptively parallelized query execution.**

time should not show much variations. In this experiment we test the robustness of the convergence algorithm in an isolated execution setting.

Graph 5.22 shows the number of convergence runs to find the optimal execution time for three invocations (experiments). Except for Q6 and Q22 all other queries show minimal variations for convergence runs. Q6 is the most simple query in the given set of queries. It shows the most speed-up amongst all queries, but that also makes it vulnerable to external factors such as operating system noise interference, etc. Since, the global minimum time is very low, even small interference affects its performance. Q22 is a complex query where join operator is always the most expensive operator.

Graph 5.23 shows the run where the global minimum time occurs during three invocations of the query set under evaluation. Depending on the resource contention and the run-time interference from the operating system we get small variations across different runs for all queries. The highest difference is observed between the first and the third run for Q19. However, overall the number of runs do not show much deviations.

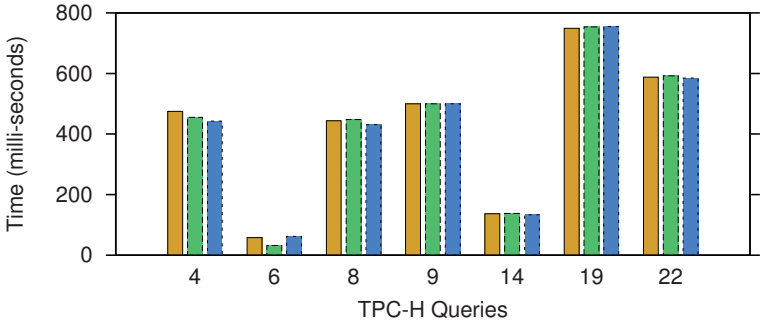

Figure 5.24: **Global minimum time for adaptively parallelized query execution.**

Graph 5.24 shows the global minimum time for adaptively parallelized queries for three invocations. The global minimum time for all queries is almost stable across multiple invocations. This indicates the robustness of the generated plans.

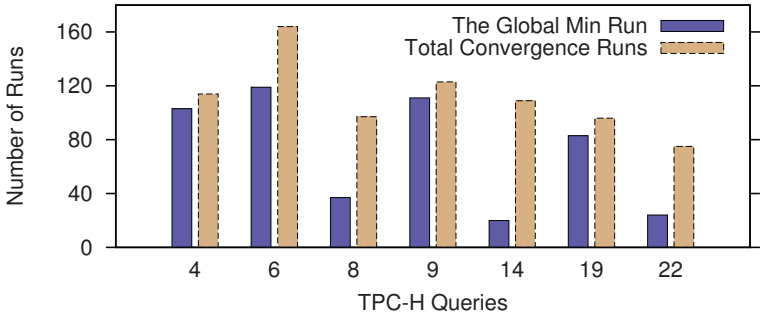

Figure 5.25: **Difference between the global minimum run and convergence runs, for adaptively parallelized query executions.**

Graph 5.25 shows the difference between the run at which global minimum occurs vs the total convergence runs. Different types of queries show different convergence properties and the algorithm gets tuned to converge in least possible number of runs. For example, Q8, Q14, and Q22 show the global minimum at less than 40 runs, while the total convergence runs are around 100. The slow convergence is a result of the *Leaking_Debit* being too low, which leads to the *credit* getting drained slowly. The convergence runs are close to 60 for the same global minimum, when the *Leaking_Debit* is high.

How to have lower number of convergence runs? As we do plan parallelization by introducing one operator per invocation, the number of convergence runs tend to be on the higher side. They can be made much lower if even number of operators

are introduced per invocation. We avoid it at present to analyze the parallel plan evolution with each new operator addition.

5.5.4 Adaptive parallelization and pipe-lined execution

Most analytical systems use columnar storage with vectorized pipe-lined execution [143, 42, 102]. Adaptive parallelization could turn out to be much simpler in pipe-lined execution, as plans tend to be much simpler with very few operators unlike MonetDB plans. As the base operators such as select become expensive, its cloned operators working on disjoint data partitions are introduced in its place. An exchange union operator combines the cloned operator's result. As operators such as the exchange union operator become expensive they get pushed up in the plan, further parallelizing the dependent operators. If an operator in the middle of the plan such as the join becomes expensive, a hash based partition operator first partitions its input and new cloned operators work on the partitioned data. The convergence is also expected to be much faster due to much less number of operators unlike MonetDB plans.

5.5.5 Future work

As MonetDB uses an intermediate language (MAL) for plan representation, the plans tend to be complex (See Figure 5.7) and their dynamic manipulation is non-trivial. It reflects in many of the decisions we make. A promising approach is plan level parallelization, based on learning from the historical plans. At present the input plan is considered to be near optimal with a heuristic based join ordering. We intend to explore the effect of feedback on join orderings. For the feasibility scope we restrict ourselves by invoking the same queries, however, another interesting aspect is to reuse the plan sub-templates to optimize similar queries. As MonetDB plans have complex operator dependencies (Figure 5.7), it makes pattern identification for adaptive plan aware query re-write non-trivial. We intend to be able to generalize the process to get more queries working in the future.

5.6 Related work and applicability to other systems

The basic optimizer approach of "optimize once and execute many" as proposed by System R has reached its limits [134]. Hence, adaptive query processing techniques are being proposed to address query optimization problems due to unreliable cardinality estimates, data skew, parameterized query execution, changing workload, complex queries with many tables, etc. [58]. In this section we describe some state-of-the-art adaptive techniques.

Adaptive aggregation is used by the authors in [49] to handle different group-by based parallelization cases. The operator performs a lightweight sampling of the input to choose the best aggregation strategy with high accuracy, at run-time. Algorithmic approaches are based on using independent and shared hash tables with locking and atomic primitives to minimize hash table access contention. Three cases are identified that affect performance based on the average run-length of

identical group-by values, locality of references to the aggregation hash table, and frequency of repeated access to the same hash table location. This work targets adaptivity from a single operator's perspective, whereas our work targets it at the plan level. The approach used here can be combined with our adaptive approach to improve per operator performance.

Vectorwise uses micro-adaptivity to improve query execution time by using run-time execution feedback [128]. Micro-adaptivity is defined as the ability to choose the most promising execution primitive at run-time, based on real time statistics. Most methods, like adaptive parallelization, use plan level modification, whereas micro-adaptivity uses the available execution primitives at run time. It chooses primitives based on the platform, instance, and call adaptivity using parameters such as compiler, branch prediction, selectivity, loop unrolling, etc.

The Learning Optimizer (LEO) in **DB2** uses query execution feedback for cardinality estimation corrections [138]. It uses learning and feedback based infrastructure to monitor query execution and generates feedback for correction in the cardinality estimation and related statistics. More learning helps in better cost model predictions. LEO has improved query execution performance by orders of magnitude. MonetDB does not use cardinality related statistics, however if used with the statistics correction methods, the selection of operator's to parallelize can be improved further.

In [34] authors illustrate adaptive parallel execution in Oracle for big data analytics. In Oracle problems such as reliance on query optimizer estimates are handled by changing the data distribution decisions adaptively, during query execution.

Column store architectures differ in various aspects such as plan representation, partitioning strategy, scale out support, etc. Encompassing all the requirements in a single architecture is not possible due to their architectural confinements. While describing the related state-of-the-art column stores, we also describe the possibility of adaptive parallelization's application to them.

Vertica uses a value based partitioning approach [99]. It uses a Read Optimized Store (ROS), where the data is stored in multiple ROS containers on a standard file system. Two files per column within a ROS container are stored, one with the actual column data and the other with position index. This representation is very similar to the representation in Figure 5.10, where RH is the index, while RT is the actual value. Vertica also supports grouping multiple columns together in a file, however this hybrid row-column storage is rarely used in practice because of the performance and compression penalty it incurs.

Vertica execution engine uses a multi-threaded pipe-lined vectorized execution where the execution plan consists of standard relational algebra operators. Operators such as StorageUnion are used for partitioning data across operators. Hence, StorageUnion is equivalent to a partition operator. Operators such as ParallelUnion are used for directing execution to multiple threads and to combine the parallelized operator's results. Hence, ParallelUnion is equivalent to an exchange union operator.

To understand the feasibility of applying adaptive parallelization in Vertica,

let's assume that the execution starts with a serial plan and incrementally introduces value based partitions to partition the expensive operator's data. For example, when a select operator becomes expensive and needs to be parallelized, a partition operator is introduced which creates two value based partitions, which would be consumed by two new select operators. The two new select operator's output is combined using an exchange union operator. This is similar to the basic mutation scheme with the addition of a partition operator that feeds two newly introduced select operators.

When one of the newly introduced select operators itself becomes expensive, we further partition that operator's data into two more value based partitions, by introducing a new partition operator in the data flow path after the previous partition operator, and before the input of the expensive select operator. Thus in a hypothetical case when one of the select operators stays expensive during consecutive invocations, new partition operators would keep on getting added to the existing plan. We expect the cost of the partitioning operator to be small considering its presence in the existing Vertica execution plans. Quantifying the exact cost is difficult due to lack of sufficient references. Similar logic can be applied for other operator's parallelization.

Apollo creates column store indexes in a traditional row store database like SQL server [102]. It is the first database which uses the existing row store to create new column store indexes. The method involves creation of batches of rows to create segments from which individual columns are stored in individual column representations. The column segments information is stored in the directory structure, with a catalog.

The columns are compressed and encoded using different types of encoding. New operators called batch operators are introduced which get called if there is bulk data to be processed. The valid rows to process are noted down in bitvector formats.

Apollo uses range partitioning of data. Since traditional SQL server uses cost model based exchange operator induced parallelization, Apollo leverages the existing SQL server parallelization technique using the exchange operator based parallelization.

To understand how adaptive parallelization might be applied in Apollo type of column store, we need to find similarities between adaptive parallelization and Apollo architecture. Both do range partitioning of data, hence the fundamental assumption of range partitioned access stays the same, and could change in the way individual operators are implemented. For example, the operations like the join operator consists of separate build and probe operators, where build uses a shared hash table, where all threads build a hash table, and then probe operator probes it in parallel. As Apollo extends the exchange operator based parallelization as used in SQL server, we expect adaptive parallelization to be useful, due to its dependence on the exchange operator based parallelization.

Hyper uses LLVM [103] generated Just In Time (JIT) compiled plans. The longest pipeline in a plan is identified, by looking for a pipeline breaker operator. The operators in the longest pipeline are fused using JIT compilation such that their highly

efficient machine language code represents a single task. The fusing allows tuples to be kept in registers to process them without generating intermediate results. Hyper's morsel driven parallelism uses work stealing based approach to assign the fused pipeline tasks to a fixed number of pre-created threads. The task allocation based approach allows controlling the number of tasks executing in parallel dynamically, at run-time, and allows better control over resource allocation during concurrent execution of queries.

Adaptive parallelization technique is based on the fundamental assumption that an expensive operator is always identifiable in an execution plan. This is a basic requirement since plan parallelization is a result of incrementally parallelizing the expensive operator during successive query invocations, until a global minimum plan is identified.

Identification of a single expensive operator is not feasible in Hyper's execution plans due to the JIT compiled fused nature of operator's pipeline, which prevents a direct application of adaptive parallelization. However, in a broader sense if the entire task is considered to be expensive and treated as an expensive operator, application of the adaptive parallelization logic can be possible. Hence, the feasibility of adaptive parallelization technique in Hyper depends on how to categorize the expensiveness metric.

DB2 BLU accelerator [124] uses evaluator chains, which comprises DB2 BLU operators working on columnar data. The data is accessed in strides. It uses novel data structures that minimize latching allowing seamless scaling with multi-cores. Parallelism involves cloning of evaluator chains once per thread, where the number of threads is decided by cardinality estimates, system resources and system load. Each thread requests strides for its evaluator chains until no more strides are available. DB2 BLU also uses work stealing based approach where worker threads operate on tasks comprising of evaluator chain based work.

5.7 Summary

The research question: In this Chapter we address the questions, "How to leverage multi-core systems to improve the performance of analytical workloads?" and "What is the effect of multi-core hardware on the effectiveness of the query optimizers?"

Identifying an optimal degree of parallelization and an optimal parallel execution plan is a difficult problem as the number of CPU cores increase. Allocating all cores to the parallelized query does not result in improved performance. It could also lead to degradation due to overheads in management of the extra resources. Cost model based parallel plans are very sensitive to operator cardinalities, hence do not generate optimal parallel plans either. Identifying an optimal parallel plan is thus a challenging problem.

The research contribution: We introduced *adaptive parallelization*, a feedback based technique to identify near optimal degree of parallelization and multi-core utilization in a parallelized query execution environment. It is inspired by the ob-

servation that most analytical queries are template based. When the same query gets fired multiple times, adaptive parallelization comes into effect. During each invocation the most expensive operator in the query plan is incrementally parallelized, until a good parallel plan is identified.

We introduced a convergence algorithm that stops the feedback loop when a good enough parallel plan is identified. Using extensive experimentation we have shown that adaptive parallelization generates better plans. We have shown that the convergence algorithm is robust in identifying a good plan. An important finding is how adaptive parallelization improves multi-core utilization which helps during concurrent workload execution as it uses less resources due to less number of data partitions compared to heuristic based plan parallelization. Adaptive parallelization has generated interest from industry leaders such as the Oracle Labs, the HP Labs, and IBM research. It has inspired further research from research groups such as the University of Athens, Federal university of Parana, Brazil, University of Luxembourg, and others. In Section 5.6 we summarize how it could be used in existing database systems that use the *exchange* operator based parallelization by doing suitable adaptations.

5.8 Conclusion

Adaptive parallelization uses query execution feedback to generate resource contention and skew aware range partitioned multi-core parallel plans. It helps in finding the right balance between the multi-core utilization and the degree of parallelism for the exchange operator based parallel plans. We observe a near linear speed-up with the number of cores while analyzing the parallel plan evolution using parameters such as the partition range, the input size, and the selectivity. During TPC-DS isolated workload, the adaptively parallelized plans show up-to five times better performance compared to heuristically parallelized plans. During TPC-H concurrent workload, they show minimal multi-core utilization, allowing better resource utilization. They also fare competitively with work stealing based scheduling approach.

Using different convergence scenarios we show that the adaptive parallelization convergence algorithm behaves robustly, and converges in a reasonable number of runs.

Chapter 6

Multi-core column store parallelization under concurrent workload

"Everyone has a plan until they've been hit." – Joe Lewis

Columnar database systems are designed for an optimal OLAP workload performance. They strive for maximum multi-core utilization under concurrent query executions. However, most of them generate a multi-core parallel plan in isolation, which during concurrent query execution leads to sub-optimal performance. To get better insights into how resource contention affects individual multi-core query performance, there is a need to analyze its effects on the intra-query and inter-query parallelized plans.

In this Chapter [1], we analyze the concurrent workload resource contention effects on multi-core plans using three intra-query parallelization techniques, *static*, *adaptive*, and *cost* model parallelization. We focus on a plan level comparison of selected TPC-H queries, using in-memory multi-core columnar systems. Excessive partitions in statically parallelized plans result in heavy L3 cache misses leading to memory contention, degrading query performance severely. The operating system's default scheduling policy influences the degree of parallelism of static plans. Overall, adaptive plans show more robustness, less scheduling overheads, and an average 50% execution time improvement compared to statically parallelized plans, and cost model based plans.

[1] This Chapter is based on the publication "Multi-core column-store parallelization under concurrent workload", Under submission.

6.1 Motivation

The ubiquitous presence of multi-core CPUs calls for an analysis of their optimal utilization by database systems, under OLAP workloads [130, 113]. Most systems use either intra-query or inter-query parallelization to maximize multi-core utilization. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing. Inter-query parallelization involves executing individual queries on each core, as used by e.g. Postgres [139, 13]. Intra-query parallelization involves parallelization of a query plan using the *exchange* operator, to execute on the available cores, as introduced by the Volcano system [70], and used in most commercial systems. An issue ignored in most cases is that the performance of an individual query is strongly affected by the presence of a concurrent OLAP workload, which leads to resource contention, as queries compete for shared resources such as CPU, memory, and IO bandwidth [24]. Higher resource contention leads to extended query execution times, thereby increasing the multi-core utilization, and in turn decreasing the overall query throughput [24]. A simple solution that can be deployed is to limit the degree of parallelism of plans.

As run time resource contention is difficult to model, static and cost model based approaches can not consider it during plan generation. Hence, quantifying the effect of a concurrent workload on an individual query's performance is difficult [144, 76]. One of the fundamental approaches is to use workload variation models to analyze their resource contention effect on a sequential query performance [60, 25]. The resource contention problem becomes even trickier to handle during parallel plan execution, as the contention could negate the gains due to parallelism, and make identification of the degree of parallelism difficult. To gain better insights we categorize different types of workloads in a broad manner based on *inter-query* or *intra-query* parallelization modes, and analyze how the resource contention affects an individual parallelized query's performance.

We evaluate three types of intra-query parallel plan generation techniques, *static* [64], *adaptive* [2], and *cost model* [20], under concurrent OLAP workloads, using in-memory multi-core columnar database systems. Static parallelization involves row-id based range partitioning, without accounting for the resource contention. Adaptive parallelization is a new feedback based plan generation technique that performs incremental query parallelization, since many workloads use template based repeated queries. Repeated query invocations introduce more partitions in an already parallelized plan, until a plan with minimum execution time is identified. When adaptive parallel plan generation happens in the presence of a concurrent workload, it reflects the impact of resource contention. While our focus is on the static and the adaptive technique, we also compare both these techniques with a cost model based intra-query parallel plan generation technique.

6.2 Contributions

Most database systems generate an optimal parallel plan without taking into account the run-time resource contention, as modeling run-time resource contention is very difficult. However, it leads to the plans behaving sub-optimally under concurrent workload. There is not much research literature giving insights into the effect of concurrent workload execution on parallelized query execution. We analyze the effect of three intra-query parallelization techniques in the context of two columnar database systems, and also compare it with inter-query parallelization technique. Our main contributions are as follows.

- We evaluate the performance of intra-query parallel plans, under different types of in-memory concurrent workloads. More data partitions in *static plans* result in heavy L3 cache misses leading to memory contention. *Adaptive plans* with less data partitions show up to 50% better performance. *Cost model plans* with admission control result in severely degraded performance.

- We provide a categorization of workloads based on their average CPU core idleness (which reflects their multi-core utilization), when the concurrent workload server executes in either inter-query or intra-query parallel mode. We show that even a broad workload categorization as we do, helps in getting critical insights into performance behavior of parallelized queries under analysis.

- We show that a workload with random queries generates less resource contention than a workload with similar queries, when the execution engine does not have support for intermediate data sharing.

- Finding robust query execution plans is a critical requirement during workload analysis. We analyze the robustness of parallelized plans under concurrent workloads, where the select operator plans exhibit more robustness than the join operator plans. Overall adaptive plans show more robustness than static plans. On the other hand, inter-query parallelization as used by Postgres shows degraded performance compared to the column stores, however, its overall behavior is more robust.

- We highlight the influence of the operating system's default thread scheduling policy on the degree of parallelization of parallelized plans.

Observations from this research can help in mixing workloads such that the resource contention due to them is not as bad as some workload combinations that we illustrate. The experimentation provides good insights into two types of parallelization techniques (intra-query parallelization and inter-query parallelization), in the context of columnar database systems. The generic observations from them such as restricting the number of data partitions to a minimum during parallelization could be applicable to other columnar systems as well, to minimize resource contention.

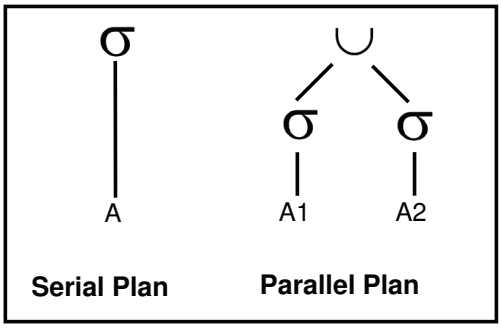

Figure 6.1: **Serial and Parallel plan.**

6.2.1 Outline

The Chapter is structured as follows. In Section 6.3 we provide a brief background of the static, cost model, and adaptive parallelization techniques. In Section 6.4 we describe the set-up for the concurrent workload to generate resource contention. We provide a detailed experimental analysis to understand the effect of resource contention in Section 6.5. Related work is described in Section 6.6. In Section 6.7 we conclude summarizing the main lessons learned.

6.3 Plan parallelization techniques

Multi-core parallel plan generation using the exchange operator [70] is a two stage process in most database systems. During the first stage an optimal serial plan is generated, while in the following stage partitions are introduced using the exchange operator, to generate a multi-core aware parallel plan.

We use two exchange operator based columnar systems, MonetDB the open-source operator-at-a-time execution system and Vectorwise, a leading analytic system with pipe-lined vectorized execution. Being the only open-source columnar system, both, static and adaptive parallelization techniques are implemented in MonetDB. Plans are represented using an *abstract intermediate language* called MonetDB Assembly Language (MAL) [40], with operator's implementation in "C". Vectorwise on the other hand uses a cost model based parallel plan generation. Plans use normal physical algebra operator representations. A dedicated buffer manager with predictive buffer management and co-operative scan support helps in concurrent workload disk IO sharing. MonetDB on the other hand relies on the operating system managed memory mapped buffers.

6.3.1 Static parallelization

Static parallelization **(SP)** already exists, and is the *default* parallelization technique in MonetDB. It is done in two steps. First, the *largest* table in the plan is partitioned such that the number of partitions equal the number of

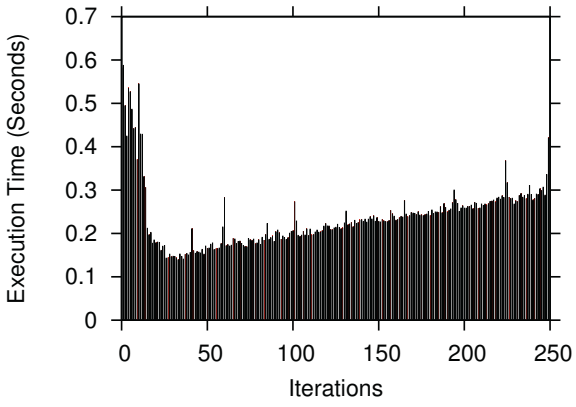

Figure 6.2: **An adaptively parallelized plan execution sequence for TPC-H Q14.**

cores [64]. Next, the operators in the data flow dependent path of the partitioned base operator's data are parallelized such that the data flow pattern is maintained.

In Figure 6.1, Plan 1 shows a simple serial plan with the selection on A. Plan 2 shows the parallel plan derived by row-id based equi-range partitioning of A, with two new select operators. An *exchange union operator* (U) combines the parallelized operator's result. Based on the query complexity, the plans could have complex dependency patterns, and multiple exchange union operators. Aggregation is postponed as much as possible. Static parallelization is simple and fast, but can be less accurate than the cost model based parallelization.

6.3.2 Cost model based parallelization

The cost model based plan generation in Vectorwise involves predicting the cost of execution of a plan using an operator's input type and estimates about its cardinalities, to choose supposedly the optimal plan. Size of the data is one of the biggest deciding factors, but memory hierarchy and access pattern, processor characteristics, and concurrent query execution affects the overall prediction considerably. Parallel plan generation uses the exchange operator based partitioning of an optimal input serial plan. A combination of branch and prune and dynamic programming techniques are used in the search strategy for the parallel plan. An elaborate discussion on the related topics is in [20][4][147].

Both static and cost model based techniques are sub-optimal since the run-time resource variation can not be accounted for. In contrast, adaptive parallelization takes into account resource contention through execution feedback.

6.3.3 Adaptive parallelization

Adaptive parallelization **(AP)** [2] is a new parallelization technique developed in MonetDB, inspired by the observation that most systems use a relatively small

number of parameterized query templates repeatedly. While the technique is completely described in [2], we provide a brief overview here.

AP uses execution feedback to incrementally parallelize a query plan with each successive query invocation. A parallel plan (P1) is generated from a previous plan (P0), by parallelizing the most expensive operator from P0. The AP infrastructure stores previously executed plans along with the profiled information such as the operator execution time and resource claims. Under concurrent workload the execution feedback reflects the resource contention making adaptive parallel plans more robust compared to SP plans, as we show during the experiments.

Figure 6.2 shows an AP plan execution sequence in action where the X axis represents the consecutive invocations (iterations) of the same query. In the current setup, for the feasibility scope purpose, we repeatedly use the same query, though most systems re-use query templates. Each vertical bar in the graph represents the plan execution time corresponding to a particular invocation. The 0th invocation corresponds to a serial plan execution, while the 1st invocation corresponds to the plan derived by parallelizing the most expensive operator from the 0th invocation plan. With consecutive invocations more operators in consecutive plans get parallelized leading to an execution time improvement until a global minimum is reached (the 19th iteration). More parallelization afterwards leads to a performance degradation. In an ideal scenario each successive plan provides better performance than its predecessor. In practice the execution skew due to introduction of only two partitions in successive iterations prevents it, and prolongs the convergence.

Convergence: Depending on the query complexity the number of iterations taken by AP to converge could vary. For the ease of the experimental set-up we use a fixed 250 iterations as it covers all possible query convergences. The convergence algorithm and various convergence scenarios are discussed in detail in [2]. For example, TPC-H Q14 shows minimal execution time at the 19th iteration (See Figure 6.2). However, the convergence runs for other queries could vary between 50 to 100 iterations.

Global Minimum: The speed-up of a plan is measured with respect to its serial execution. We term a plan as the global minimum plan if its speed-up is better than all other minimal plans observed earlier during the global minimum search. We run multiple experiments and consider the average of the global minimum plan's wall clock time.

6.4 Workload set-up

In this section we describe the concurrent workload set-up to generate resource contention for shared resources such as the CPU cores, to analyze its impact on a parallelized query. We use batched OLAP queries to generate the workload. The number of concurrent OLAP queries in a batch decides the multi-programming level (MPL) of the system. First we elaborate on the client setup, then the workload server setup, and next the query set used in the workload from MonetDB and Vectorwise perspective.

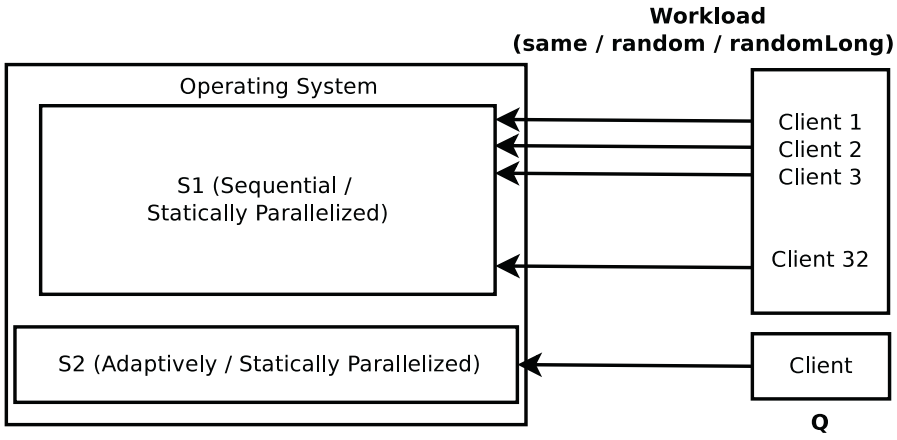

Figure 6.3: **The workload set-up.**

6.4.1 Client setup

The concurrent workload consists of 32 clients (MPL = 32) connected to a MonetDB execution instance (S1 in Figure 6.3). As our experimental platform uses 32 CPU cores (Hyper-threaded), the number of clients are limited to 32 to ensure each CPU core has at least 1 connection. We do not aim to test the *scalability* aspect at present. The clients repeatedly fire TPC-H queries (scale factor 10) from one of the three query mix batches as shown in Table 6.1. The intention is not to measure *throughput*, but to keep the system always busy. The *long* queries execute in more than 1 second, where the slowest query executes in around 10 seconds. In contrast the *short* queries execute in less than 1 second.

Table 6.1: **Query mix batches.**

B1	B2	B3
Same	Random	RandomLong

All queries in the batch B1 are the same, and match the parallelized query under the resource contention impact analysis. The batch B2 has a mix of eight short and eight long queries, and the batch B3 has ten long queries. There can be many other possible batch configurations [55], however the aim here is to show that the broad workload categorization also provides good insights into the resource contention effect on individual parallelized queries. For example, when clients use queries from B1, the aim is to imitate the scenario when the concurrent workload consists of queries that work on similar data.

6.4.2 Server setup

We use two MonetDB execution instances S1 and S2 (See Figure 6.3), for the ease of experimental setup. The MonetDB execution instance **(S1)** for concurrent client connections is executed in either sequential **(Inter-query)** or statically parallelized **(Intra-query)** mode.

Table 6.2: **% CPU core idleness for *M*onetDB and *V*ectorwise workloads(To be read as - ServerExecutionMode_QueryMix). Note.* - Different queries have different CPU core idleness, hence not shown.**

Seq_Same	Seq_Random	Seq_RandomLong	Par_Same	Par_Random	Par_RandomLong
15 % ***(M)*** ***(V)***	22% ***(M)*** 27% ***(V)***	26% ***(M)*** 12.6% ***(V)***	0% ***(M)*** ***(V)***	13%***(M)*** 27% ***(V)***	0%***(M)*** 10.6% ***(V)***

Table 6.3: **System configuration**

CPU	Sockets / Threads	L1 / L2	Shared L3 cache	Mem	OS
Intel Xeon E5-2650@ 2.00GHz	2 / 32	32 / 256 KB	20MB	256GB	Fedora 20
Intel Xeon E5-4657Lv2@ 2.40GHz	4 / 96	32 / 256KB	30MB	1TB	Fedora 20

In sequential (SQ) mode, S1 executes a query per core such that with 32 clients, 32 queries execute on 32 cores (hyper-threaded), leading to inter-query parallelization. With this set-up we try to imitate the scenario where some database systems try to maximize multi-core utilization by executing a single query per core [139].

In statically parallelized (SP) mode, S1 does an intra-query partitioned parallel execution, such that depending on the number of row-id based range partitions, a query could get parallelized to execute on all the available cores. As a result during the concurrent workload of 32 clients, 32 queries execute in SP mode on all the available cores.

Depending on S1's execution mode and the query mix type (B1 / B2 / B3), **6 workloads** are possible as listed in **Table 6.2**. Our aim is to analyze the resource contention effect of these workloads on a single query's (Q) parallelized performance.

To achieve that we use a dedicated MonetDB instance **(S2)** (See Figure 6.3) to execute Q in AP or SP mode, in the presence of the concurrent workload executing on S1. For an AP execution a client connected to S2 repeatedly executes the same query Q for 250 iterations. Both AP and SP execution in S2 works on in-memory data without any disk IO (hot runs).

Why use separate S1 and S2 instances? Separate instances of the servers S1 and S2 allows us better instrumentation abilities for measuring the hardware events for the parallelized query (Q) under analysis. The MonetDB profiler does not use per client based connection, but has a global view of the entire execution engine. Separating S1 and S2 instances allows isolating Q's profiler statistics from the statistics of the 32 concurrent queries.

Separate execution instances however does not affect the resource contention impact from S1 on S2, as the resources such as caches, memory, and CPU cores are shared globally. MonetDB *does not* use a dedicated buffer manager, but uses a memory mapped based buffer management infrastructure. Hence, the operating system handles buffer management, thread scheduling etc. at the holistic system level.

6.4.3 Query set (Q)

MonetDB query plans tend to be complex due to data flow dependencies of multiple operators, represented in MonetDB Assembly Language (MAL). Hence, we identify a subset of TPC-H queries (scale factor 10) to support adaptive parallelization,

to analyze the concurrent workload's resource contention effect on them. The scale factor 10 is used as it provides sufficient insights about the resource contention effect. Our attempts to use higher scale factor such as SF-100 resulted in heavy disk IO, due to large intermediates in MonetDB. The current implementation supports a single group-by attribute queries. Hence, we modify some of the existing TPC-H queries to suit this need. Since the performance of both SP and AP is analyzed using the same set of queries, the comparison is fair.

Table 6.4: **Query set (Q) categorization**

Simple	Q6	Q14			
Complex	Q4	Q8	Q9	Q19	Q22

6.4.4 Vectorwise setup

During Vectorwise experiments we use a single database instance (10GB), on which 32 concurrent clients execute queries using one of the workloads as shown in Table 6.2. The query Q is invoked on the same instance unlike MonetDB, so that Vectorwise plan generation resource allocation scheme could take into account the load, in terms of the number of concurrent clients. MonetDB plan generation does not take into account the run-time clients, hence having two instances S1 and S2, does not affects its plan generation. We use Vectorwise version 3.5.1 in the default configuration with the parallelism set to 32 and the histogram statistics support enabled. We also use the clustered index.

6.4.5 Workload and CPU core idleness

Concurrent workloads have different multi-core utilization depending on the query mix, which in turn leads to varied CPU core idleness. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing [64]. The higher is the multi-core utilization the lower is the idleness. In our setup the operating system allocates idle cores to the parallelized query under analysis allowing it to progress. When not idle, CPU cores are either busy with computations or memory access. Table 6.2 lists the average CPU core idleness for different types of workloads, measured as an average of 3 batches of 20 samples at 1 second sampling period, using the *sar* command.

Table 6.2 shows when using MonetDB (M), S1 executes parallely, CPU cores are mostly busy compared to when S1 executes sequentially. This is expected as when S1 executes in parallel, many parallelized operators from multiple queries execute concurrently. Context switching leads to the operators getting scheduled on multiple cores, resulting in cache thrashing which leads to constant memory access, keeping the CPU cores constantly busy. As the sequential execution has relatively less concurrent executions, it leads to considerable less cache thrashing and less memory access, resulting in idle CPU cores.

6.5 Experiments

Table 6.3 lists our experimental hardware platform details, which consists of two types of machines, with two and four socket CPUs each. All experiments unless otherwise mentioned use a hot execution run (no disk IO) on in-memory data, on the 2 socket machine. We use the 4 socket machine in a few selected experiments to show the NUMA effect. We repeat the experiments four times and report the average. We use *perf* tools [56] to measure hardware events that reflect the resource contention impact on a subset of the queries. For the rest of the queries we use their response time as a measure to reflect the resource contention impact.

Different concurrent workloads lead to different resource contention scenarios, thereby changing the parallelized query Q's execution time. The aim is to identify robust parallel plans which get minimally affected due to resource contention under different concurrent workloads. With this fundamental goal we explore the following questions in the context of different concurrent workloads.

1. How the number of partitions influence plan parallelization ?
2. Which plans perform better and exhibit more robustness?
3. Where does time go during resource contention?
4. Which is better inter-query or intra-query parallelization?

In the rest of the section we analyze each of these questions in detail.

6.5.1 How the number of partitions influence statically parallelized plans?

As statically parallelized plans in MonetDB are relatively simple to generate, if made resource contention aware, they could offer an easy solution to the plan parallelization problem. In this section we analyze if a heuristic optimizer can generate better parallel plans under concurrent workload, by tuning parameters such as the *number of partitions*.

The optimizer controls the number of partitions by controlling the number of threads. Hence, using fewer threads lead to a different plan with fewer partitions. The hypothesis is, this plan might show better concurrent workload execution performance, as it puts less pressure on the shared resources such as the CPU cores, and the memory bandwidth due to fewer partitions. NUMA effects also could play a role. Hence, we test the hypothesis using the 2 socket and the 4 socket machine.

2 Socket NUMA

Figure 6.4a shows the MonetDB query execution times for varying degree of parallelism, when the concurrent workload = Parallel_Random on the 2 socket NUMA machine. It nullifies our earlier hypothesis as irrespective of the number of threads in use, the minimal time occurs at 16 or 32 threads, where physical cores are 16, and 32 includes hyper-threads. Similar observations are made for other type of workloads.

This behavior can be explained by the fraction of the idle CPU cores available during the concurrent workload. When workload = Parallel_Random each CPU

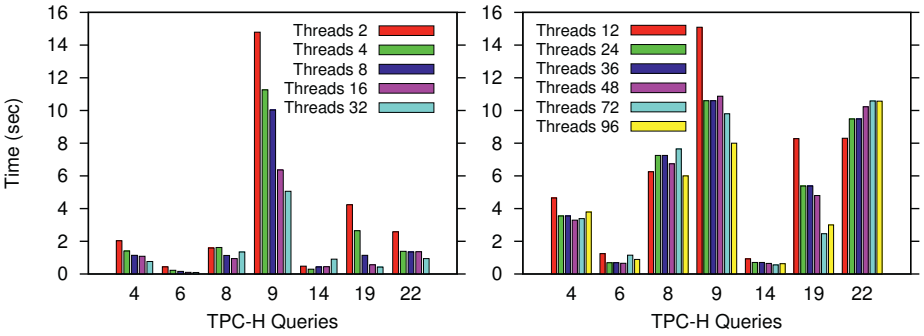

Figure 6.4: **Effect of degree of parallelism on query execution under concurrent workload a) MonetDB static parallelization on a 2 socket machine (10GB data). b) MonetDB static parallelization on a 4 socket machine (100GB data).**

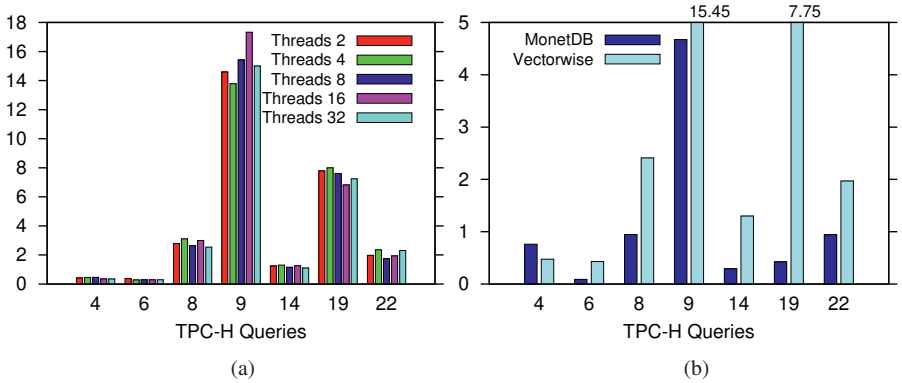

Figure 6.5: **Effect of degree of parallelism on query execution under concurrent workload a) Vectorwise cost model parallelization on a 2 socket machine (10GB data) b) Best execution time for MonetDB vs Vectorwise using execution times from 6.4a and 6.5a.**

core has an average idleness of 13% (See Table 6.2), which is available for the parallelized query to progress. The default operating system scheduling policy (CFS) ensures a **load balanced** fair share from all the CPU cores to the parallelized query. Hence, a 16 or 32 threaded execution (with 16 or 32 partitions) performs better than a fewer threaded execution. The queries which show better performance for 32 threads get benefited from the hyper-threads. Change of thread priorities in the Linux scheduler could alter this behavior, however we do not explore it, as we use out of the box settings.

4 Socket NUMA

Figure 6.4b shows that when the same experiment is repeated on the 4 socket NUMA machine on a 100GB dataset, the results are quite different. No explicit NUMA aware data partitioning is used as MonetDB uses memory mapped storage [65]. Execution with up-to 48 threads uses the physical threads (12 threads on each socket with numactl [15] process and memory affinity), whereas 72 and 96 threaded execution also uses the hyper-threads. The behavior of each query is quite different as depending on the query complexity and NUMA access, different execution pattern is observed. For most queries around 96 threads leads to the minimal execution time, except for Q22, which shows a distinct different behavior. Deciding exact number of partitions to give best execution is difficult[3], so partitions equal to total number of hyper-threaded cores can be a reasonable heuristic in the multi-socket machines.

Number of partitions and cost model plans

Figure 6.5a repeats the experiment for Vectorwise cost model based execution on the 2 socket machine with 10GB dataset. The first observation is irrespective of the number of threads the query execution time does not change much. We did not anticipate this behavior, because in an isolated execution setting with increasing number of threads we did see the query execution time improving with increasing threads up-to 8 threads, and then staying almost constant. We do not plot this graph due to the space constraints. The scaling problems beyond 8 cores in isolated execution can be explained by [20], due to exchange operator scalability, lock synchronization issues, etc. In a concurrent workload we hypothesize the following things might be happening. Vectorwise's cost model based parallelization approach takes into account the load on the system in terms of the number of clients. Hence, heavy concurrent workload leads to an almost sequential execution as only a single core gets allocated for the queries under analysis. Hence, change of number of threads does not change the execution time.

Figure 6.5b shows the best execution time obtained using varying number of threads for MonetDB is much better than the Vectorwise time, which indicates cost model plan generation using resource allocation control might not lead to best performance under heavy concurrent workload.

We saw the influence of the number of partitions on the parallelized execution. The operating system's thread scheduling policy also has an important role to play in this setup. The micro-experiment we describe next gives more insight.

CPU core idleness and OS scheduling

As the concurrent parallelized queries execute on all CPU cores, controlling each CPU core's idleness is not possible, which makes the operating system's scheduling role analysis difficult. The next micro-experiment allows us a fine grained control over each CPU core's idleness, using a concurrent workload called, *Infinite Loop*.

Infinite Loop workload: The workload consists of a CPU core hogging program such as a *while(1);* loop, executing on individual CPU cores, thereby keeping them

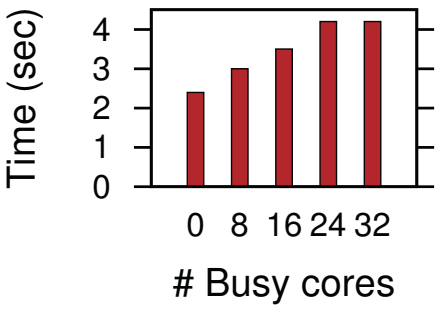

Figure 6.6: **Q9 with 100% busy cores when concurrent workload = Infinite Loop.**

100% busy. Optimization is turned off to make sure the compiler does not optimize the loop. The workload allows us to have a fine grained control over the number of busy cores at any instant. We observe the execution time of all queries on a 32 threaded statically parallelized database instance, while the Infinite Loop workload is active. (Fewer or more than 32 threads does not give better performance). We do not enumerate cores in any specific order (like logical CPU order, socket / core/ HT order), but let the operating system use its default scheduling policy. Figure 6.6 plots the execution time (Y-axis) for query 9, while varying the number of 100% busy CPU cores (X-axis). Query 9 is the longest running query in Q, hence is expected to show the largest performance variations with CPU resource share variations.

Figure 6.6 shows as the number of exclusive busy cores are increased, Q9 execution time increases by around 0.6 seconds. Though the cores are made 100% busy in a stepped manner, the operating system does ensure some quanta of resources on even the busy cores. Hence, the busy cores also contribute towards the parallel query execution. The idle cores contribution depends on the type of the query (CPU / memory bound). When 24 cores are made busy, we observe that the operating system changes its scheduling policy and does load balancing such that now all the cores are busy. However, the cores are not 100% busy, thereby introducing some idleness on each of them.

When 32 cores are made 100% busy, since there are no more spare resources available, we do see all cores 100% busy again. However, the share of CPU resources allocated to Q9's execution does not change after the 24 busy core case. Hence, the query execution time does not change when 32 cores are made 100% busy. Some other queries do show an increase in execution time when all 32 cores are made busy. We also overload each of the CPU cores with multiple CPU core hogging programs to observe its effect on the query execution time. However, we get similar results as shown in Figure 6.6.

Summary: Overall we observe that the operating system ensures a load balanced fair CPU resource share guarantee on a 2 socket machine. It ensures the best result is obtained when the number of threads equals physical cores (16) or the number of hardware contexts (32). In a NUMA setting depending on the query complexity,

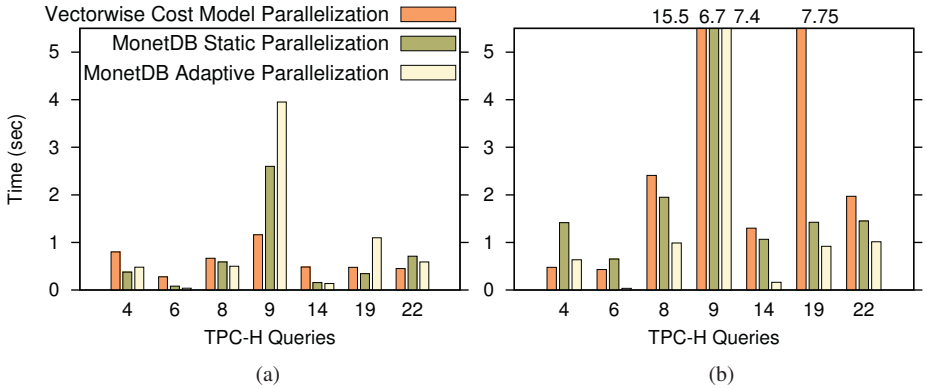

Figure 6.7: **a) Isolated execution. b)Query execution when concurrent workloads are Parallel_Random.**

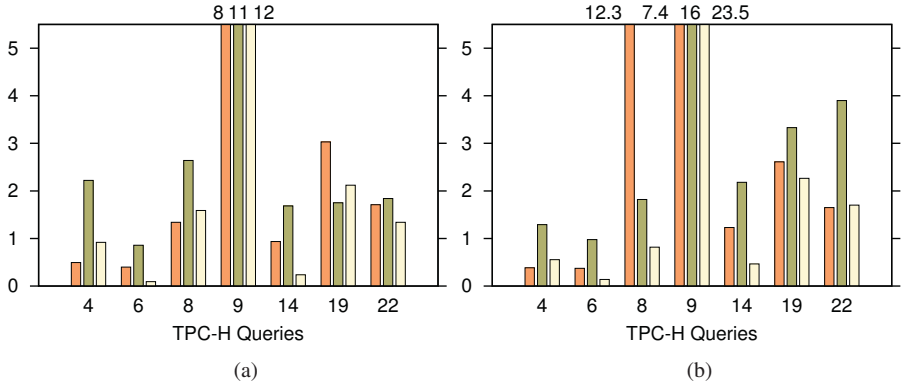

Figure 6.8: **Query execution when concurrent workloads are a) Parallel_RandomLong b) Parallel_Same.**

different execution times are observed, largely due to remote accesses. However, for many queries when partitions equals either physical cores (48) or the number of hardware context (96), the best execution time is observed. Overall, hyper-threads benefit some queries. Vectorwise shows scalability issues beyond 8 threads, but performs reasonably compared to MonetDB.

6.5.2 Which is better, static, adaptive or cost model parallelization?

In this section we analyze how different plans compare from execution performance and robustness perspective. We showed that in MonetDB static plans with partitions equal to number of hardware context provides the best execution time during concurrent workload. However, these plans need not offer good performance due more resource consumption, since all parallelizable operators in it use a fixed degree of parallelism. In contrast adaptive parallelized (AP) plan's operators are

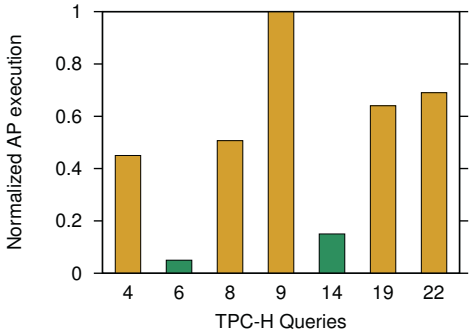

Figure 6.9: **Adaptive parallelized execution normalized with statically parallelized execution when concurrent workload = Parallel_Random.**

parallelized individually using the execution feedback, thereby allowing each parallelizable operator to have a different degree of parallelism, which helps during concurrent workload. We expect the cost model plans to show performance in between static and adaptive plans due to degree of parallelism decision based on the cost model.

Isolated execution: As a baseline reference we use *isolated* execution comparison (See Figure 6.7a), where a single query executes in the system without any concurrent workload. Most queries do not show much improvement in AP compared to SP and cost model plans. As a worst case Q19 even shows a much degraded performance in AP. It results from the presence of some non-parallelizable operators. On the other hand Q9 takes much more time in MonetDB static parallelization due to multiple joins in relatively large table attributes, as only lineitem table is partitioned in static parallelization. Parallelization using partitioning in isolation can improve performance only up-to a threshold, which can be seen, since all techniques show a comparable performance, except a few. Rest depends on how optimal plans are in terms of resource consumption, which is where AP fares better under concurrent workload. We analyze it in the rest of the section.

Performance

Adaptively parallelized plans perform better than the static and cost model based parallel plans, during concurrent workload due to optimal multi-core utilization, about which we illustrate next.

Figure 6.9 gives an overview of the performance gains in AP compared to SP when the concurrent workload = Parallel_Random. For better readability the AP execution time is normalized with respect to SP, such that, when SP execution of Q19 is 1 second, the AP execution is 0.6 seconds. While the simple queries benefit the most as is evident from Q6 and Q14 which show around 90% improvement (Green), on an average AP shows 50% improvement compared to SP for most queries. To gain better insights about individual query's performance we do an operator level analysis of some of these query plans, using Figure 6.7 to 6.8.

Query 8: Figure 6.7b shows when workload = Parallel_Random, Q8 performs

better in AP than in SP. The query plan contains *join* as the most dominant operator. Plan analysis shows that the number of tuple reconstruction and join operators in the SP plan are an order of magnitude more compared to the AP plan. In many column stores tuple reconstruction operators are implemented as join operators, so they do random look-ups. Too many join and tuple reconstruction operators executing concurrently in SP plan causes costly random memory access, and leads to memory bandwidth pressure. AP plan performs better as it has much less number of join and tuple reconstruction operators. Vectorwise plans execute sequentially under heavy concurrent workload, as its plan generation logic uses the number of active clients to decide CPU core resource allocation for the newly admitted queries.

Query 19: Figure 6.7b shows when workload = Parallel_Random, Q19 appears to perform better in AP than in SP. However, a comparison with AP from the isolated execution (See Figure 6.7a) shows AP execution timings does not change much. AP performance looks better as SP execution is three times expensive compared to its isolated execution.

The cause is that Q19 has the *select* operators as the dominant operators, whose parallelization during AP invocations results in the addition of new *exchange union* operators for combining their results. Based on the input selectivity the exchange union operator becomes an expensive operator after a few invocations, and gets pushed higher in the plan. However, the data flow dependencies due to a system specific non-parallelizable operator does not allow it and prevents further parallelization of Q19 plan. The SP plan does not faces this problem due to its use of the static partitions, which ensures the presence of the *exchange union* operators much higher in the plan.

Though AP performance does not change under concurrent workload, SP shows a degraded performance as a result of the resource contention due to the presence of more number of operators, as was the case in Q8. We provide a detailed quantitative analysis of the resource contention effect on Q19 SP execution using a subset of hardware event measurements, in the Section 6.5.3.3.

In cost model plans Q19 benefits due to the co-operative scans based data sharing in all the workloads, hence get minimally affected, as we illustrate in the Section 6.5.3.3.

Summary: We observe that the AP plans show better response time than the SP plans, and cost model plans. The static plans (SP) have too many operators working on fixed sized partitions, which creates scheduling and resource contention overhead under concurrent workload. Since in AP the old plan is mutated into a new plan by partitioning the most expensive operator's data, only a few operators get parallelized, where the generated partitions are dynamically sized. Some AP plans could however perform lower than the SP plans due to the presence of inherently non-parallelizable operators. Cost model plans show worst performance as during resource contention they execute sequentially, due to admission control policy based plan generation.

Robustness

Robustness is the ability of the database system to perform well under a variety of conditions including adverse run-time conditions due to data volume, data skew, and resource contention [144, 72]. Our focus is on the robust query processing during resource contention arising due to shared CPU cores. We consider a query plan to be robust if it gives minimal variations in the execution time under changing run-time conditions [71]. We analyze the query execution robustness by comparing its SP / AP isolated execution against the concurrent workload execution for Parallel_Random (CPU core idleness = 13%) and Parallel_RandomLong (CPU core idleness = 0) workload. Overall, the SP plans show more rapid degradation than the AP plans during concurrent workload.

Select operator: First we compare the AP execution of the queries where the select operators are dominant. Queries 4, 6, and 19 get minimally affected when workload = Parallel_Random (See Figure 6.7b), while they slow down by around a factor of two when workload = Parallel_RandomLong (See Figure 6.8a). Select operators involve either a point select or a range select operation on sequential data. As the Parallel_Random workload has average CPU core idleness = 13%, the select operators get sufficient CPU resources to execute, compared to the Parallel_RandomLong workload which has 0% average CPU core idleness.

Join operator: During AP execution the queries 8, 9, and 22 where the join operators are dominant, execute around 2 and 3 times slower for Parallel_Random and Parallel_RandomLong workloads respectively. The join operators are expensive compared to the select operators as they do random memory access keeping the CPU cores busy. As the average core idleness changes from 13% to 0% across the two workloads, their execution degrades due to insufficient CPU resources.

During the Parallel_Random and the Parallel_RandomLong workload the SP execution of the complex queries (4, 19, 8, 9, and 22) show a slowdown of 3.5 and 5 respectively, while the simple queries (6 and 14) slowdown 7 and 10 fold. The lack of CPU resources to execute many concurrent operators and the memory bandwidth pressure due to concurrent access is one of the main reasons for their rapid performance degradation. However, as the SP queries have too many select and join operators, isolating the exact reason for the degraded performance per operator level is difficult to access.

In cost model plans we are not able to find any concrete relation between the isolated execution and the execution under Parallel_Random and Parallel_RandomLong workload. Since the execution is expected to be almost sequential due to the heavy load, the queries could show at least 16 times degraded performance compared to their isolated execution (16 is the number of physical cores). Based on our observations of isolated query executions, the cost model parallelized plans use varying degree of parallelism, unlike static parallelization in MonetDB. This gives rise to overall varying CPU core idleness compared to MonetDB for concurrent workload, thereby making the performance under different workloads less robust.

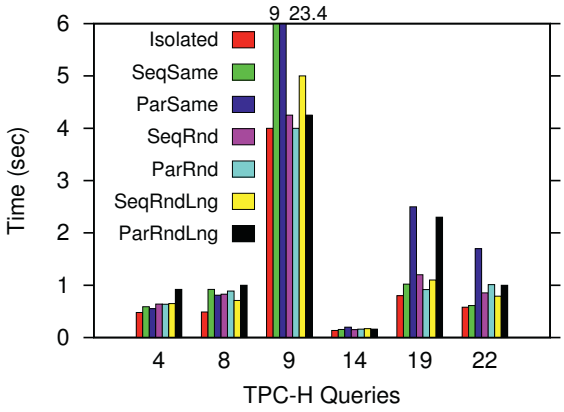

Figure 6.10: **Adaptive parallelization query performance for different concurrent workload scenarios. Table 6.2 gives the legend description.**

Query 14: AP execution of Q14 represents a special case for the Parallel_Random and the Parallel_RandomLong workload. Its plan contains a mix of both the select and the join operators as the dominant operators, unlike the other queries which we analyzed earlier in this section. The join operators however work on much less data as it gets filtered by the select operators, making them overall less expensive. The number of select and join operators in the AP plan is much less than the SP plan, as a result Q14 gets minimally affected across both the Parallel_Random and Parallel_RandomLong workload. Much less number of operators allow it to progress even in minimal CPU resources while incurring minimal memory bandwidth pressure, while exhibiting a robust behavior across the workload changes.

Summary: We observe that for AP and SP, the query execution robustness under resource contention is strongly influenced by the parameters such as the number of operators, the type of operators, and the available CPU resources. As the select operators are cheap compared to the join operators, plans where the select operators are the dominant operators show more robust behavior compared to the plans with the join operators. Overall, the AP plans are more robust than the SP plans. Cost model plan's robustness is difficult to judge from the available observations.

Having seen the performance and the robustness comparison of the parallelization techniques, next we investigate how the resource contention affects them.

6.5.3 Where does time go during resource contention?

work / data sharing

Concurrent workloads often involve queries that overlap computation and data access. Standard techniques for work / data sharing include cache (data / instruction) and operator sharing as used in StagedDB and QPipe [78, 77], multi-query opti-

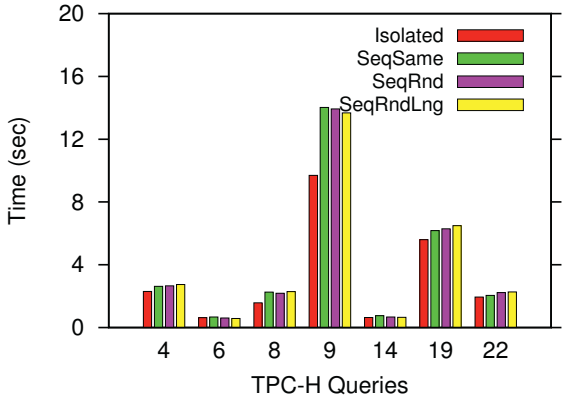

Figure 6.11: **Sequential query execution performance for different MonetDB concurrent workloads, when workload server (S1) executes in sequential mode.**

mization as used in SharedDB [66], and shared scan techniques [147]. In the absence of these techniques each concurrent query competes for the shared resources such as CPU caches (data / instruction), memory and disk IO bandwidth, etc. Since we do not employ any of the work / data sharing techniques, we do not expect individual range partitioned parallelized queries to perform better under concurrent workload. However, quantifying and analyzing the negative effects due to lack of work / data sharing still assumes importance as most database systems do not use work / data sharing techniques.

MonetDB sharing: Figure 6.10 shows how the concurrent workload *degrades* the performance of AP queries, evident from the isolated execution case (black - first bar), which shows the minimal execution time in all the workload scenarios. We do not plot query 6 as it shows minimal variations. We also do not plot the graph for SP execution as it exhibits a similar behavior, where the isolated execution has the minimal time.

Past research shows that in a batched ***serial*** query execution, depending on the query mix type, some queries get sharing benefits resulting in their performance improvement compared to their isolated execution [24]. We test it in our setup by using the concurrent sequential workloads. Figure 6.11 plots the *sequential* query's execution time when the concurrent workload server S1 also executes in a sequential mode in MonetDB. The isolated query execution shows the best execution time. Hence, in our query mix combinations the concurrent sequential workload does not help the sequential query execution.

Vectorwise sharing: Figure 6.12 shows for Vectorwise irrespective of the mode of execution (sequential / parallel), when the workload contains the same queries (SeqSame / ParSame) the query execution time is similar. This confirms our hypothesis that under heavy concurrent workload even parallelized Vectorwise queries execute near sequentially. However, execution under both SeqRnd and ParRnd workloads show more time (Q8,Q9,Q19,Q22) which indicates that some sharing opportunities might be existing when the workload contains the same queries. In order to under-

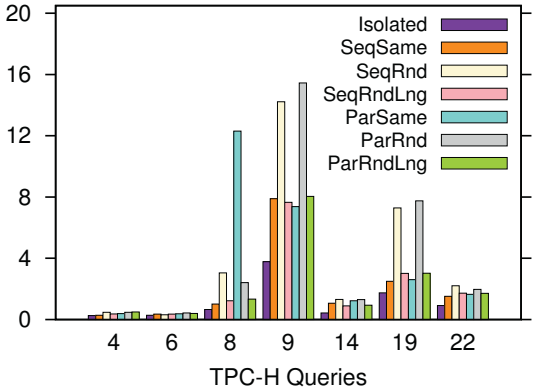

Figure 6.12: **Query execution under Vectorwise concurrent workloads.**

stand if cooperative scan technique which is designed for IO sharing can be helping in any manner at cache level sharing, we conduct a micro-experiment.

Table 6.5: **Buffer size impact on Vectorwise Q1 execution.**

Buffer size (MB)	Isolated (sec)	ParSame (Sec)
300	2.67	5.1
500	0.35	4.67
65000	0.33	4.68

Micro-experiment: Table 6.5 shows when we reduce the buffer size from the default 65GB to 300MB the isolated execution shows degraded performance. When the buffer size is increased till we get improved isolated execution performance (500MB), the performance under the concurrent workload does not increase much. Using [4] measurements as a reference, for 10GB non-compressed data-set, Q1 uses 463MB of compressed data, which fits in 500MB buffers, which explains the isolated execution performance difference for varying buffer sizes. The concurrent execution however does not show corresponding improvement, which verifies that cooperative scan technique does not help the in-memory context.

In the random workload (SeqRnd & ParRnd) there is minimum data sharing opportunity, as also confirmed from [4], where the authors show that TPC-H queries have minimum sharing opportunities. We expect the execution under RandomLong workload (SeqRndLng / ParRndLng) to take the highest time, but instead it takes the lowest time. Our discussion with Vectorwise team could not provide more insights into the overall observed behavior for Figure 6.12.

In the rest of the section we explore where does time go during resource contention.

Software / hardware level resource contention

The resource contention can be broadly classified into the software contention and the hardware contention.

The *software contention* arises due to the overheads in managing the shared resources such as the operating system scheduler, the lock contention manager, etc. We focus on the query scheduling overheads as the read only workload minimizes the lock contention. The waiting time a query spends in-order to get scheduled on 100% busy CPU cores provides the *query scheduling overhead*.

The *hardware contention* includes data sharing conflicts resulting in (data / instruction) cache thrashing, page fault handling, TLB invalidation, context switching, etc. [50] and the CPU contention conflicts resulting in pipeline invalidation, internal units access stalls, etc. [119].

The parallelized query execution time can be dissected into the query's isolated execution time, the software and the hardware contention overhead. We compare execution under Parallel_RandomLong workload and the *Infinite Loop* workload. Both workloads have 0% CPU core idleness, however differ in their work. Their difference indicates the hardware contention due to the Parallel_RandomLong workload, while the difference between execution under Infinite Loop workload and isolated execution indicates query scheduling overhead. Next we illustrate how to find software and hardware contention overheads.

Software contention overhead: The hardware contention impact of the Infinite Loop workload on a parallelized query execution is negligible. Since the instruction foot print of a while loop program is minimal, only a few CPU units such as the ones that deal with the instruction execution logic are busy during Infinite Loop workload, while the rest of them are idle. Lack of data access activity results in no cache or memory level contention. It is further confirmed from the observations in Table 6.6, which shows minimal difference in query execution hardware event measures under the Isolated execution and the Infinite Loop workload, for the SP execution of Q9.

Table 6.6: **Contention measure for Q9's statically parallelized execution under the Infinite Loop workload.**

	Isolated	Infinite Loop
L1 Miss %	6.6	6.5
L3 Miss %	66	58
Instructions/Cycle	.35	.41
StalledCycles/Instr	2.32	2.02

Since the hardware contention is negligible, the execution difference between the Infinite Loop workload and the Isolated execution indicates the *query scheduling overhead*.

Figure 6.13a shows during the SP execution the simple queries (Q6 and Q14) have minimal scheduling overhead compared to rest of the complex queries. As

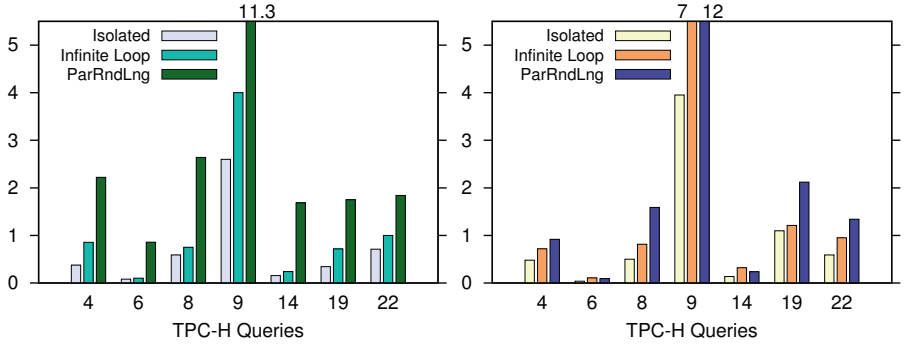

Figure 6.13: **a) The query execution time difference between Parallel_RandomLong (ParRndLng) and Infinite Loop workload reflects the resource contention impact on Statically b) Adaptively parallelized queries in MonetDB.**

the operators in the simple queries execute for a very short duration, they incur minimal scheduling overheads. The SP plans have too many operators compared to the AP plans which gets reflected in their corresponding scheduling overheads. For example, Q4 and Q19 show considerable scheduling overhead in SP execution (See Figure 6.13a) compared to their AP execution (See Figure 6.13b).

Hardware contention overhead: The hardware contention impact of the Parallel_RandomLong workload on a parallelized query execution is very high. The workload's high data access activity gives rise to heavy contention for the shared L3 cache, resulting in a large number of L3 cache misses, as can be seen in Table 6.7 for Q19's SP execution. It also results in heavy CPU level contention in terms of the high number of stalled instructions. We use Q19 to provide a perspective of the resource contention impact in terms of hardware performance events. For the other queries we use increased response time as a reflection of the resource contention impact.

In comparison the hardware level contention for the Infinite Loop workload is negligible. For simplicity we assume the query scheduling overhead for both the workloads is similar, though we expect Parallel_RandomLong workload's query scheduling overhead to be relatively more than the Infinite Loop workload's overhead, as the concurrent queries use more time quanta during their schedule. Hence, the execution difference between the Parallel_RandomLong and the Infinite Loop workload indicates the *hardware contention overhead*.

The contention overhead during AP execution (See Figure 6.13b) is much less compared to the SP execution (See Figure 6.13a) as fewer range partitioned operators execute in AP plans compared to SP plans. Fewer operators induce less scheduling overhead, and less cache thrashing.

Summary: In our concurrent workload setup, the SP / AP query under analysis does not benefit from the work / data sharing, as we do not employ explicit sharing

techniques. We segregate the contention overhead in the query scheduling overhead and the hardware contention overhead. AP plans show less scheduling overhead compared to SP plans. The *hardware contention* due to the concurrent query workload is responsible for the much degraded SP execution. L3 cache thrashing is the major source of contention in SP execution.

Having established the approximate resource contention overheads in SP and AP execution in a holistic manner, we now focus on the analysis of an individual query's SP and AP execution.

Workload specific resource contention

We analyze the workload specific resource contention effects on the query execution by comparing the Parallel_Same (Figure 6.8b) and the Parallel_RandomLong (Figure 6.8a) workloads. Since the average idleness per CPU core is zero [2] for both the workloads, one hypothesis is, the query execution time for both should be similar. However, since that is not the case, it hints at the possibility of workload specific effects on the query execution. We explain it in the context of Q19 next.

Query 19: The analysis of the SP execution of Q19 in both workloads shows the execution is two times slower for the Parallel_Same workload, compared to the Parallel_RandomLong workload. AP execution for both the workloads on the other hand does not show much variation.

The SQL level analysis of Q19 shows the *where* clause contains a union of the results of the three sub-queries. Each of the sub-query has a range based selection predicate on the same *lineitem* table attribute (Lineitem is the largest table in the TPC-H schema), with an overlapping range. The generated plan for this query takes care of maximizing sharing of the selection predicates, so that the redundant work is avoided.

When workload = Parallel_Same, since the concurrent workload involves the same query, and since the query has shared predicates, it leads to access to the same base data. Since MonetDB uses memory mapped storage, these data files get shared mapping in the memory. However, storing and loading of the intermediates as they are not shared across queries generate memory bandwidth pressure. Table 6.7 quantifies the contention impact on Q19 and provides insight for its slow down.

It shows the percentage of the L3 cache misses is very high (73%). Very high value of L3 cache misses also indicates the pressure on the memory bandwidth. The processor pipeline is heavily stalled during the cache misses resolution, resulting in its very low utilization as seen from the Instructions per Cycle and Stalled Cycles per Instruction values. Low value of instruction cache misses (79934) compared to the isolated execution (142079) indicates the instruction cache sharing. However, any gains due to it are subdued by the dominance of the L3 cache thrashing.

[2] An exception is Q4 and Q8 in Parallel_Same workload, where average idleness is 10% and not 0%. It explains why Q4 and Q8 show much less degradation compared to other queries, where average idleness is 0%.

Table 6.7: **Contention measure for Q19's statically Parallelized execution under different concurrent workloads.**

	Isolated	ParRandLng	ParSame
L1 Miss %	11	15	18
L3 Miss %	4	46	73
Instructions/Cycle	1.16	.21	.16
StalledCycles/Instr	.5	4.12	5.76
iCache Misses	142079	86415	79934

In comparison when the workload = Parallel_RandomLong, the concurrent workload contains a mix of queries accessing different base tables. The workload also has multiple instances of Q17. Q17 contains a selection predicate on the same attribute of the *lineitem* table as in Q19. Hence, we expect some possible sharing at the memory mapped level. In [4] authors show a matrix of TPC-H query sharing, where Q19 shares maximum data with other queries. Due to the random workload the intermediates generated are however of different sizes unlike the Parallel_Same workload thereby generating less L3 cache misses (46%), leading to a better performance. CPU utilization is also relatively more compared to the Parallel_Same workload.

Summary: Depending on the workload and the query type queries exhibit different sharing patterns, such as possible instruction cache sharing. However, no data cache sharing is observed as stores and loads of the non-shared intermediates lead to heavy last level cache thrashing, thereby degrading the SP execution performance. Q19 shows heavy L3 cache thrashing with up-to 73% L3 cache misses, resulting in its performance degradation during the Parallel_Same workload.

Vectorwise resource contention

Figure 6.14 shows that for most queries the parallelized Vectorwise query execution under Parallel_Random workload (Yellow) is on an average six times slower than the isolated parallelized query execution (Grey). The performance degradation under concurrent workload results due to resource contention and the resource allocation scheme in Vectorwise, where the queries get resources such as CPU cores based on the existing system load. The first query gets all the available CPU cores and the subsequent queries get less cores based on a certain heuristic. We hypothesize that in the existing scenario where the concurrent workload consists of continuously executing 32 queries, the degree of parallelism for the single query under analysis gets restricted to one, making its execution sequential.

To verify it we plot the sequential query execution in an isolated setting (Dark Green), which shows around two times speed-up compared to the concurrent workload execution. The execution time difference indicates the possible resource contention due to the concurrent workload on a sequential query execution. We also plot the parallelized query execution under an Infinite Loop workload executing on all CPU cores, to get an indication of the scheduling overhead for the parallelized

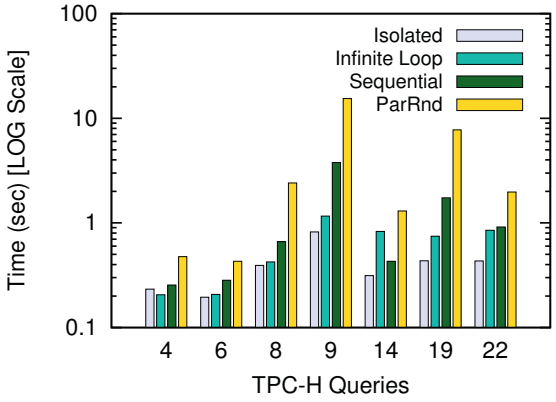

Figure 6.14: **Parallelized query execution (Yellow) performance under concurrent workload (Parallel_Random) in Vectorwise database system degrades by around 6 times compared to the parallel execution in Isolated setting (Grey). The Y axis uses a log scale.**

query. The query execution time degrades by around two times compared to the isolated parallelized execution due to the scheduling overhead.

Summary: Vectorwise parallel query execution under heavy concurrent workload of random TPC-H queries shows a degradation by around six times compared to the isolated parallelized execution. In a similar setup the MonetDB queries show a slow down by around three times. The observations suggest that a hard core heuristic on admission control as used by Vectorwise need not be always optimal under a heavy concurrent workload.

6.5.4 Which is better, inter-query or intra-query?

Systems such as Postgres [139] maximize multi-core utilization by executing a single query per core. Since multiple queries execute concurrently, queries execute in the inter-query parallelization mode. On the other hand most systems such as MonetDB, Vectorwise, Tableau, and SQL Server[43, 23, 19] use intra-query parallelization, using the *exchange operator* [70], where a single query could execute on more than one core. Use of an appropriate technique is mostly driven by the system architecture in use.

Setup: In this experiment we compare the inter-query parallelization performance of Postgres with Vectorwise and MonetDB on 10GB data-set on the two socket machine. Both Vectorwise and MonetDB are used in sequential execution to serve the 32 concurrent clients firing random queries (SeqRnd workload). The query Q under analysis is also executed in the sequential mode. We use Postgres version 9.4 and configure the parameters such as shared buffer size using pgtune [16] tool recommendations. Postgres forks 32 server processes to serve the 32 concurrent clients

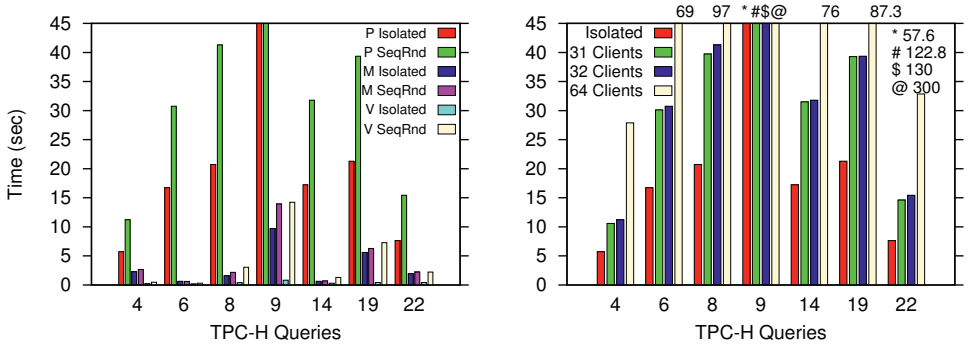

Figure 6.15: **a) Inter-query parallelism comparison of Postgres, MonetDB and Vectorwise for isolated sequential execution and for sequential execution under SeqRnd workload for 32 clients. (P- Postgres, M- MonetDB, V- Vectorwise). b) Postgres execution performance when number of concurrent clients under SeqRnd workload are 31, 32, and 64.**

firing continuous random queries. The client that fires query Q under analysis thus becomes the 33rd concurrent connection.

Performance: Figure 6.15a plots the execution performance of queries when executed in isolation vs under concurrent workload execution (SeqRnd), for the three database systems (P- Postgres, M- MonetDB, V- Vectorwise). Postgres performance in both isolated and under concurrent workload is always much lower than the corresponding MonetDB or Vectorwise performance. The much degraded performance of Postgres isolated execution is a result of its tuple-at-a-time execution engine architecture, which is not optimized for in-memory execution, unlike MonetDB and Vectorwise. An interesting observation is under concurrent workload all Postgres queries always show around two times degradation than its isolated execution. This indicates that though Postgres shows much degraded performance, still its queries show a relatively robust behavior under concurrent workload execution.

Robustness: To test Postgres robustness behavior further, we conduct another experiment where 31 and 64 clients fire random queries under SeqRnd workload. When 31 clients fire random queries, the query Q under analysis is fired by the 32nd client. The aim of this experiment is to understand when resource of one core is available, whether the query Q gets full core for its execution and behaves similar to isolated execution, since all the 31 clients are busy with 31 cores. However, the results from Figure 6.15b do not indicate that. The execution performance of queries when 31 clients are active is slightly better than when 32 clients are active. This indicates, possible sharing of 31 cores among 31 available clients due to lack of explicit core affinity, which prevents a dedicated single core allocation to 32nd client for the query Q under analysis. When 32 clients are active we see the CPU core idleness to be always 0%, while when only 31 concurrent clients are active we do see some idleness across random CPU cores.

The execution time of queries increase by two times compared to isolated execution, for 32 concurrent client workload (SeqRnd). On the other hand, for 64 clients it increases by around 4 times, except for Q9. This verifies the earlier claim that Postgres query execution under heavy concurrent workload is relatively robust, as it shows linearly degraded performance for almost all queries, with increasing number of clients.

Overall, both MonetDB and Vectorwise inter-query execution shows between 5 to 30 times better performance than Postgres under concurrent workload. In contrast the adaptive parallelized execution of MonetDB under Parallel_Random workload shows between 10 to 50 times improvement compared to Postgres execution under concurrent workload.

Summary: Inter-query parallelization as used by Postgres shows much degraded performance than inter-query sequential execution by columnar systems such as MonetDB and Vectorwise. Postgres suffers due to tuple-at-a-time execution architecture.

Overall, intra-query parallelized execution under parallel workload provides better performance than inter-query sequential execution under sequential workload. Best performance is obtained when the parallel workload consists of a mix of *random* queries, as the concurrent random query execution does not stress the memory bandwidth. MonetDB's plan generation without explicit resource control fares better than Vectorwise, which uses concurrent client connections, as a reference during plan generation.

6.6 Related work

The related work is categorized into two broad levels. The first one is query interactions in concurrent workload using model based approaches. The second one is analyzing the resource contention effect of concurrent workloads in the multi-core CPU setting.

A lot of past work deals with identifying the correct multi-programming level (MPL) and aligned problems. MPL decides maximum number of queries that can simultaneously execute in a system without degrading the overall system performance. Many times a scheduling based approach is used to model different possible query mix interactions, as used in [26]. In [140] the authors use an admission control and model based approach to decide the most suitable MPL. Other research includes work by [25, 55, 98, 97, 96, 112] to model the query interactions and concurrent workload effect on individual OLAP queries. Feedback based approach in transactional workloads, as used in [133] also gives overall idea of possible technique that can be applied in OLAP workloads.

Identifying the resource contention effect on an individual query performance has been explored in the context of sequential query execution, in the context of pipe-lined parallelism. Pipe-lined parallelism minimizes intermediate data unlike the operator-at-a-time execution. Authors in [90] explore the trade-offs of work sharing vs pipe-lined parallelism in multi-core systems in sequential query execution. Simultaneous threads (SMT) on multi-core processors can be considered as contributing to a concurrent workload. In [146] the authors investigate the effect

of SMT on database workloads. In [76] the authors do a thorough analysis of contention in chip multi-processors at different CPU cache levels. However, none of this work explores the problem of resource contention in range partitioned parallelism in an in-memory muti-core setting. One of the reasons being partitioning is difficult to set-up and can suffer from changing data as re-partitioning is not easy.

state-of-the-art systems such as Hyper [104] use morsel driven work stealing based run-time adaptive parallelism. In Hyper terminology, controlling the number of partitions is equivalent to controlling the size of a morsel. It allows the degree of parallelism variations of an individual query elastically. However, a direct comparison under heavy concurrent workload with adaptive parallelism is not feasible, due to lack of availability of Hyper for our experimental setup.

In [67] the authors propose a new mechanism to minimize resource utilization and to maximize performance and predictability while deploying query plans on multi-core systems. They propose resource activity vectors to characterize individual database operator's behavior. A new deployment algorithm uses these vectors with data-flow information from the query plan for the optimal assignment of the relational operators to the cores. In [44] the authors introduce a new scheduling mechanism for multi-core systems where instead of CPU core oriented scheduling focus, they propose on-chip memory focused scheduling. The threads are scheduled across cores based on their data objects usage of the on-chip memory. In [79] the authors propose Callisto, a resource management layer for parallel run-time systems. The authors illustrate how Callisto eliminates most of the scheduler-related interference between concurrent jobs, and allows jobs to claim otherwise-idle cores.

Applicability to exchange operator based systems: Most systems use the exchange operator based parallelism approach where number of partitions is the main parallelization decision metric [43, 19, 23, 39]. Hence, our observations about performance and robustness with respect to the number of partitions and operators are directly applicable. Unless explicit work / data sharing techniques are used the sharing opportunities are minimal, reflecting our observations for resource contention. Many systems use customized schedulers, making the software level contention overheads vary based on specific implementation.

6.7 Summary

The research question: In this Chapter we address the questions, "What is the effect of multi-core hardware on the effectiveness of the query optimizers?" and "How to provide insights into the query execution performance bottlenecks at a database system's functionality level?"

Most database systems do not take into account run-time resource contention during plan generation. Modeling run-time resource variations is very difficult which makes the generated plans sub-optimal under concurrent workload. Investigating the effect of a concurrent workload on a parallelized query execution is crucial to make progress in creating resource contention aware parallel plans. Different types of concurrent workloads generate different types of resource contention for CPU cores, memory bandwidth, etc. Also the effect of resource contention on

a parallelized query execution varies significantly depending on the query parallelization technique (intra-query / inter-query) under use.

The research contribution: We introduce 6 different types of workloads that create different levels of resource contention at CPU cores and memory bandwidth level. We analyze the effects of these concurrent workloads on individual parallelized query execution using three query parallelization techniques in the context of three database systems. We provide detailed insights from thread variations, scheduling overheads, robustness of individual operator's, and intra-query versus inter-query parallelization perspective. We also quantify the performance effects in terms of microarchitecture hardware counters such as cache misses, pipeline stalls, etc. Analyzing resource contention due to concurrent workload is very challenging due to experimental setup complications, constant workload variations, ability to isolate individualized query performance effects, etc. This is the only work so far known to us that explores concurrent analytical workload effects on individual parallelized query execution in the different areas mentioned, in the context of three full-fledged database systems.

The insights obtained can be used 1) to design new concurrent workloads and query combinations with minimal resource contention, 2) to decide between intra-query and inter-query parallelization on the basis of performance robustness, 3) to understand why adaptive parallelization performs better than heuristic based parallelization, etc. One of the key findings is less number of data partitions lead to better resource utilization, which can be used while taking decisions during query optimization phase combined with statistics based approaches such as cost model optimization. Unless the database system has data sharing abilities, using workloads that have similar queries does not help, as these workloads generate more resource contention compared to workloads with random queries. Most database systems do not use data sharing abilities, so it should help during the workload selection.

6.8 Conclusion

Getting insights into the concurrent workload effects on a parallelized query execution is a very important problem in the field of query parallelization research. The research in this Chapter investigates it using state-of-the-art systems and provides some critical insights, such as the role of the number of partitions of a parallelized plan in reducing overall resource contention. We compared three intra-query parallelization techniques, static, adaptive and cost model based parallelization, under different concurrent workloads, using two in-memory multi-core columnar systems. We show that even a broad categorization of concurrent workloads, co-related with average CPU core idleness as a metrics gives good insights into parallelized query execution performance. We show that random queries based workloads generate less resource contention compared to workloads that contain similar queries, when intermediate data sharing is not supported by the execution engine.

Adaptive plans show overall better performance during resource contention as

they generate less partitions compared to static plans. The static plans show minimal time when the number of partitions equals the physical cores. Adaptively parallelized plans show more robustness and an execution time improvement of an average 50% compared to the statically parallelized plans. Too many partitions in statically parallelized plans leads to severe resource contention amongst the competing threads, resulting in a large number of L3 cache misses, resulting in the memory bandwidth contention. Cost model based parallelization shows the highest time as the queries are allocated minimal CPU cores due to heavy concurrent workload. An important finding is the behavior of inter-query parallelization from robustness perspective, where the inter-query parallelization shows a much robust behavior compared to the intra-query parallelization.

Chapter 7

NUMA obliviousness through memory mapping

"The world around me was oblivious, but for once, I felt absolute" – Rebecah McManus

With the rise of multi-socket multi-core CPUs a lot of effort is being put into how to best exploit their abundant CPU power. In a shared memory setting the multi-socket CPUs are equipped with their own memory module, and access memory modules across sockets in a non-uniform access pattern (NUMA). Memory access across socket is relatively expensive compared to memory access within a socket. One of the common solutions to minimize across socket memory access is to partition the data, such that the data affinity is maintained per socket.

In this Chapter [1] we explore the role of memory mapped storage to provide transparent data access in a NUMA environment, without the need of explicit data partitioning. We compare the performance of a database engine in a distributed setting in a multi-socket environment, with a database engine in a NUMA oblivious setting. We show that though the operating system tries to keep the data affinity to local sockets, a significant remote memory access still occurs, as the number of threads increase. Hence, setting explicit process and memory affinity results in a robust execution in NUMA oblivious plans. We use micro-experiments and SQL queries from the TPC-H benchmark to provide an in-depth experimental exploration of the landscape, in a four socket Intel machine.

7.1 Motivation

Most low end servers are equipped with two socket CPUs. In contrast, most high end servers tend to have four or eight socket CPUs, in a shared memory setting.

[1] This Chapter is based on the publication "NUMA obliviousness through memory mapping", In Proceedings of DaMoN, SIGMOD 2015.

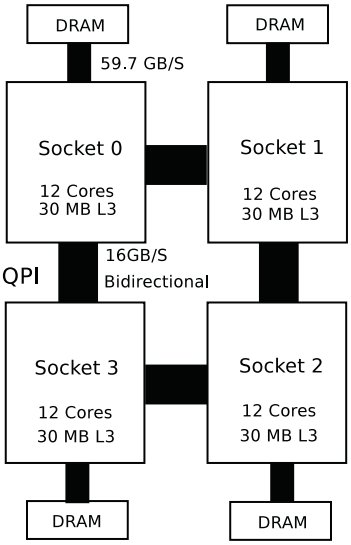

Figure 7.1: **Schematic diagram for Intel Xeon E5-4657LV2 @2.40GHz CPU**

The memory access latency in a two socket CPU is relatively low, however, in four and eight socket CPUs it is considerably expensive, if the memory being accessed is remote.

Figure 7.1 shows a shared memory four socket CPU system where each CPU socket is associated with its own memory module (DRAM), and can also access a remote DRAM through Quick Path Interconnect (QPI) [9]. The memory access latency thus varies considerably based on whether the memory being accessed is local or remote. For example, the process residing on socket 0 accesses its local memory much faster than the remote memory on socket 2, as socket 2 is 2 hops away from socket 0. Non-uniform memory access (NUMA) [100] is thus a result of different memory access latency across sockets, in a shared memory system.

The graph in Figure 7.2 shows such an example for TPC-H Q1 (Scale factor 100 GB on four socket CPU). We plot an average of 6 runs (minimal variations are observed between consecutive runs), clearing the buffer cache between independent query executions. The database server process (using memory mapped storage) is allowed to execute strictly only on two sockets (0 and 1), by pinning the process's affinity to both sockets, using the tool *numactl* [15]. On the other hand, the memory allocation for the process is allowed to take place on different sockets (0 to 3), using numactl's memory binding option, to emphasize that the locality of data and the memory access distance matters, and affects the execution time.

When the memory allocation is local (socket 0 and 1), the execution time is lowest, as there is minimal cross socket data access. The execution time is highest when the memory allocation occurs only on socket 2, as memory on socket 2 is 2 hops away from socket 0, and 1 hop away from socket 1. The operating system does not allocate memory pages in an uniform manner across sockets, hence the

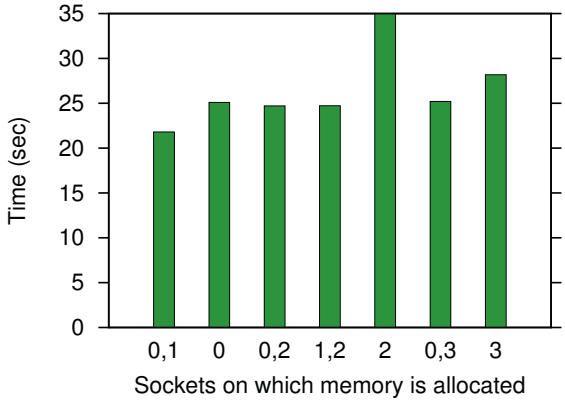

Figure 7.2: **Response time variations for TPC-H Q1 (100GB) on a 4 socket CPU, when the database server process is spawned across both sockets 0 & 1, while the memory allocation is varied between sockets 0 to 3.)**

part of the process executing on socket 0 tends to access more pages, compared to process execution on socket 1. This also explains why the execution is second highest when memory allocation occurs only on socket 3. Memory on socket 3 is only one hop away from process on socket 0, compared to the process on socket 1. From Figure 7.2 the execution time for the rest of the memory allocation affinities is more or less similar.

Database systems try to mitigate the data affinity problem in NUMA configuration by partitioning the data across CPU sockets either using range or hash partitioning[104]. For example, in a star schema, while the fact tables are horizontally range partitioned, the dimension tables being relatively small are replicated. As an example consider a 100 GB data set, where the data is horizontally partitioned in 25 GB piece each, affiliated with the sockets by introducing corresponding query plan partitions. The thread affinity is set to the corresponding sockets. This design however requires query plan level changes to introduce data location aware partitions in the plan, to maintain the data affinity to the sockets.

The observations from Figure 7.2 show minimal variations except for socket2. This motivates us to explore the viability of an alternate approach using memory mapped storage, to minimize the need for explicit data partitioning across sockets. Memory mapped IO uses the operating system's virtual memory infrastructure to control mapping of disk files to the memory. Memory mapped IO ensures that only the portion of the file gets loaded when its access is required. For example, during execution of a binary file the first step by the operating system is to do memory mapping of the disk file. As and when page faults occur, the corresponding portion from the file is brought to the memory based on the mapping. The same logic can be used to load data from disk files into memory such that locality with respect to sockets is maintained. Hence, we expect through memory mapped storage the operating system could offer an oblivious access to the data, while maintaining its locality with sockets, in a NUMA setting [14].

Consider the case of columnar database systems [110, 43, 99], where memory

mapped storage can be used to represent in memory columnar data, backed on disk by a suitable file representation. During in memory data access the data is brought to the memory from disk and stays in memory as long as there is no need to swap it out. As an example of a possible mapping, consider the case when two select operators work on a column that is partitioned into two equal halves. If the first operator is scheduled to execute on the socket 0, then its data is mapped onto the memory module for the socket 0. Whereas, if the second operator's execution is scheduled on socket 1, then its data gets mapped on the memory module for the socket 1. The operating system thus tries its best to keep the data affinity to sockets, depending on the source of access.

7.2 Contributions

Most legacy database systems are designed without taking into account the non-uniform memory access on multi-socket systems. Making these systems NUMA aware requires changes in their system architecture, which might not be practical due to code legacy and the amount of engineering effort involved. We investigate how the NUMA problems could be mitigated in the context of traditional database systems using support from the operating system features such as memory mapping. We also propose a simple shared nothing architecture to make current systems NUMA aware. Our main contributions are as follows.

1. We investigate the behavior of different types of query plans (NUMA oblivious vs NUMA aware) under NUMA settings. We show that the NUMA oblivious query plans using memory mapping feature, provide reasonable performance compared to NUMA aware partitioned plans.

2. We investigate the effect of memory mapping in the context of NUMA setting and provide insights into the behavior of memory mapped columnar storage.

3. We show that remote memory accesses lead to performance degradation in NUMA oblivious plans. To minimize remote memory accesses, we propose a simple NUMA aware architecture that can be used by existing legacy systems without changing their architecture. In this architecture a multi-socket system is configured as a shared nothing database system, resulting in minimal remote memory accesses, improving the execution performance by up to 3 times.

7.2.1 Outline

The Chapter is structured as follows. In Section 7.3 we briefly describe the NUMA oblivious and NUMA aware plans. In Section 7.4 we provide the experiments to analyze the memory mapped IO behavior in a NUMA setting. In Section 7.5 we provide a perspective compared to a leading database system. Section 7.6 describes the related work. We conclude in Section 7.7 citing major lessons learned.

7.3 NUMA oblivious vs NUMA aware plans

Columnar database systems are a good experimental platform since the columnar storage can be represented in a memory mapped file. MonetDB, the only open-source columnar database system is a good choice, since it uses a memory mapped columnar storage for base tables and intermediate data. It uses operator-at-a-time execution model, completely materializing the intermediate results.

We use three separate configurations to test the effect of NUMA oblivious data partitioning vs NUMA aware data partitioning. We describe these configurations next.

NUMA oblivious data partitioning: MonetDB uses a simple heuristic such that a parallel plan is generated from a serial plan by range partitioning the largest table in the plan. The number of equi-range partitions equal the number of the available cores. Operators in MonetDB plans operate on the range partitioned data, where they get scheduled on the available cores using the default operating system scheduling policy (CFS). This scheme represents multi-core intra-query parallelism, where data partitioning is done at plan level, without explicit socket knowledge. The operating system takes care of scheduling the operators on the sockets such that the memory affinity is maintained in a NUMA setting [100, 14]. This scheme thus does not involve any kind of explicit NUMA related optimization with respect to explicit horizontal data partitioning, and hence is termed as **NUMA_Obliv**.

NUMA aware data partitioning: To explore the effect of socket aware partitioned data access we use a modified implementation of MonetDB tailored towards the socket based data locality. The data is partitioned horizontally in 4 pieces such that the *lineitem* and the *orders* table are partitioned across sockets, while the rest of the tables are replicated. This modified implementation of MonetDB uses an optimizer that generates socket aware partitioned plans. Inspired by [131] we use MonetDB in a distributed master slave architecture. We name this implementation **NUMA_Distr**.

We assign one MonetDB server instance per socket which acts in a slave configuration, whereas a Master MonetDB server instance executes on any one of the four sockets. Thus we have a total of five MonetDB server instances, one of which is a master and the rest four are slaves. Slaves execute when master is not executing, hence the presence of a separate master does not involve resource sharing. The slaves carry out the execution of partitioned plan corresponding to their partitioned data, while the master is responsible for the final aggregation of individual results from each of the four slaves. Each one of the slaves in turn operates on an intra-query partitioned plan where maximum partitions are the number of cores per socket. The intra-query partitioned plans that each one of the slave uses are generated using the same NUMA oblivious parallel plan generation logic. Thus the NUMA_Distr mechanism essentially limits the access of data locally and prevents across socket interference.

We also use another variation of plans which are similar to NUMA_Distr plans in their physical representation, however in their execution behavior are similar to

NUMA_Obliv. In this scheme a single MonetDB instance uses horizontally partitioned data (lineitem and orders tables) across four sockets. The parallel plans generated in this manner are socket aware, however since we do not use any kind of thread binding across sockets, the operating system is free to schedule the threads based on its default scheduling policy, thereby making them behave in a NUMA_Obliv configuration. We name this mechanism as **NUMA_Shard**, because it works on sharded data like in NUMA_Distr scheme, however without the master slave configuration. This configuration is used to overcome the partitioning problems in NUMA_Obliv configuration, that arises due to lack of partitions on the orders table. In Section 3.1 we elaborate it using TPC-H Q4 as an example.

Summary: Note that all these three configurations use memory mapped storage, as MonetDB uses memory mapped files to store columnar base and intermediate data. Though in NUMA_Distr separate database servers execute on each socket, the individual operators in the plan work on the memory mapped stored data, restricted to each socket. Hence, execution performance comparison of these techniques reflect the effect of memory mapping.

7.4 Experiments

The hardware comprises of Intel Xeon E5-4657L v2 @2.40GHz with 4 sockets, 12 cores per socket for a total of 96 threads (Hyperthreading enabled), L1 cache=32KB, L2 cache=256KB, and shared L3=30MB, and 1TB four channel DDR3 memory where each socket is attached to 256 GB memory. The operating system is Fedora 20. The results are an average of 6 runs. The buffer cache is cleared [2] between successive query executions to allocate new memory mapped pages, to avoid interference from previously pinned pages. As MonetDB uses memory mapped columnar storage for base and intermediate data, memory mapping is always enabled for all three configurations, NUMA_Obliv, NUMA_Shard, and NUMA_Distr.

We use the tools *numactl* and *Intel PCM* to get insights into the effects of local vs remote memory accesses.

Numactl: We use numactl [15] to control the process and memory allocation affinity to individual sockets. An example command to set the database server process affinity to sockets 0,1 and memory affinity to socket 2 is as follows.

```
numactl -N 0,1 -m 2 database_server_process
```

Intel PCM: We use Intel Performance Counter Monitor (PCM) tool [8] to measure the CPU performance events. PCM is different from frameworks such as PAPI [114] and Linux Perf [56] because it not only supports *core* but also *uncore* events. The uncore is the part of the processor that deals with integrated memory controllers, the Intel QuickPath interconnect, and the IO hub. We use the executable *pcm-numa* to measure the local and remote DRAM access, of cache-line size unit. Linux Perf [56] tool is used to measure the CPU migrations.

[2] echo 3 — sudo /usr/bin/tee /proc/sys/vm/drop_caches

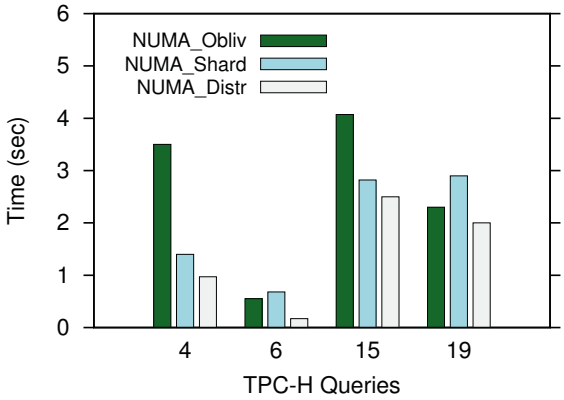

Figure 7.3: **Query execution performance of NUMA oblivious vs NUMA aware partitioned plans, for scale factor 100.**

7.4.1 SQL query analysis

We use a subset of SQL queries from the TPC-H benchmark on a 100 GB dataset, to analyze their execution performance in NUMA oblivious vs NUMA aware partitioned plans. We then switch to micro-benchmark queries for a fine grained analysis of the observations from the SQL queries.

Setup: NUMA_Obliv setup uses a single instance of MonetDB with varying number of threads, executing on 96 cores, with default operating system scheduling policy (CFS). The NUMA aware plan execution setup (NUMA_Distr) uses four instances of MonetDB with 24 threads each, bound to each one of the four sockets, using *numactl* tool. The client invokes queries on a separate MonetDB instance which acts as a master. The other NUMA aware plan setup (NUMA_Shard) also uses a single instance of MonetDB with varying threads, on 96 cores, with sharded lineitem and orders table. The dimension tables are not replicated.

Figure 7.3 shows query execution time comparison for selected TPC-H queries. We use this query set as it provides sufficient insights into the overall behavior of the techniques under comparison.

The first observation is NUMA_Distr shows the best execution time in all the queries. This is expected because of minimum cross socket interference due to master slave configuration.

Next we analyze individual queries by focusing on Q6 first, where NUMA_Distr shows around 3 times improvement compared to the other two configurations. Both NUMA_Obliv and NUMA_Shard show similar times. Q6 has a single lineitem table with only select operations, which get parallelized easily. The difference in execution with NUMA_Distr is due to cross socket traffic, as both
NUMA_Obliv and NUMA_Shard does not have explicit data affinity in plans, as their threads get scheduled according to the default operating system scheduling policy.

NUMA_Obliv shows highest time for Q4, due to MonetDB's parallel plan gen-

eration limitation. Q4 has both the lineitem and the orders table. Orders table is the second largest table in TPC-H after lineitem table. As MonetDB optimizer partitions only the largest table for generating a parallel plan, only the lineitem table is partitioned. Q4 has a join on the lineitem and the orders table attribute, which contributes to the lengthy execution as the orders table is not partitioned. In both NUMA_Shard and NUMA_Distr versions both the lineitem and the orders table are partitioned, which explains why both of these configuration are much faster.

Query 19 illustrates the effect on a more complex query over two tables, the lineitem and the part table. However, in NUMA Shard only the lineitem table is being partitioned into four pieces using row-id ranges. The part table is not partitioned. This implies that all cores involved in the join will randomly access the part table, increasing the intra-core memory accesses. A hash-based partitioning could alleviate this overhead, but is currently not part of the MonetDB standard repertoire.

As observed from Q6 and Q19 we expect NUMA_Obliv configuration to be competitive to NUMA_Distr configuration provided majority of the tables in the plan are correctly partitioned. The NUMA_Shard configuration is used just to prove this point, since otherwise as can be seen in Q4, NUMA_Obliv execution looks too expensive. For a drilled down analysis of the effect of the memory mapped IO in a NUMA setting, we focus on Q6.

Why focus on Q6? Q6 is a simple query with select operation on the largest table, lineitem. Parallelized select operators are expected to execute with memory affinity maintained to sockets due to memory mapped storage. Hence, analyzing it gives a baseline to analyze the numa effects. We hypothesize that the difference in timings for Q6 (See Figure 7.3) is due to cross socket interference. This is confirmed from Table 7.1 which shows that NUMA_Distr has much less number of remote memory accesses compared to NUMA_Obliv. Hence, we investigate where do these remote memory accesses arrive and if they can be curtailed to improve the NUMA_Obliv execution further.

Table 7.1: **Q6 memory accesses (cache line size unit).**

	#Local accesses	#Remote accesses
NUMA_Obliv	69 Million (M)	136 M
NUMA_Distr	196 M	9 M

7.4.2 Micro-experiments

We use a modified Q6 from the TPC-H benchmark. Q6 operates on the largest table *lineitem*. The query is modified to have only a single select operation, without any output. It allows us to experiment with the read only aspect of the memory mapped IO. The query is as follows.

select count(*) from lineitem where l_quantity >24000000;

The select operator acts as a good example to demonstrate the effects of memory mapped IO in a NUMA setting. It is easily parallelizable by range partitioning of the data, such that each partition is operated upon by one select operator. Since each partition uses a memory mapped storage representation, we hypothesize that the operating system would schedule select operators on sockets, such that the data affinity is maintained, resulting in NUMA obliviousness.

To test our hypothesis we control the socket allocation for memory and database server process execution. The 4 socket CPU has 12 physical cores and 12 hyper-threads per socket, in the following order.

Table 7.2: **CPU core allocation across sockets.**

	Socket 0	Socket 1	Socket 2	Socket 3
Cores	0-11	12-23	24-35	36-47
Cores	48-59	60-71	72-83	84-95

Execution with numactl affinity setting

Setup: The graph in Figure 7.4a quantifies the remote vs local memory accesses, when process execution and memory allocation affinity is set using numactl. The process affinity is set on the sockets in steps of 12 threads, as per the core order in Table 7.2. The memory affinity to sockets is also allocated in increments of one socket each. For example, when 12 threads execute on socket 0, the memory allocation is also pinned to socket 0, whereas when 36 threads execute on three sockets (as per core order in Table 7.2), the memory allocation is pinned to three sockets. The corresponding command for 36 threads is as follows.

numactl -C 0-11,12-23,24-35 -m 0,1,2 Database_Server

First observation in Figure 7.4a is when 12 threads execute on the socket0, their remote memory access is almost negligible, and the entire memory access arrives from the local memory. As the number of threads increase across the sockets, the local memory access decreases, while the remote memory access increases until 60 threads are in use. Note that in Table 7.2, hyper-threads form the range 48-95. Hence, 60 threads on-wards both remote and local memory accesses almost stabilizes. Figure 7.4b, shows the corresponding execution time, which also almost stabilizes after 60 threads of execution.

Figure 7.5 shows the proportion of pages mapped on each socket as the number of sockets increase. Consider the case, when 24 threads execute on the socket0 and the socket1. Since only around 30% pages are mapped on the socket0, almost two third of the threads executing on the socket0 do remote memory access to access pages from the socket1, rest do local access. Since socket1 has all its pages mapped, we expect all threads on socket1 to do local access. This indicates the remote access is around 1/3rd of the total page accesses, while the local accesses are 4/6th of the total accesses. However, the numbers from the 24 thread case, in Figure 7.4a does not reflect it. Remote accesses are higher, than local accesses. Digging deeper in

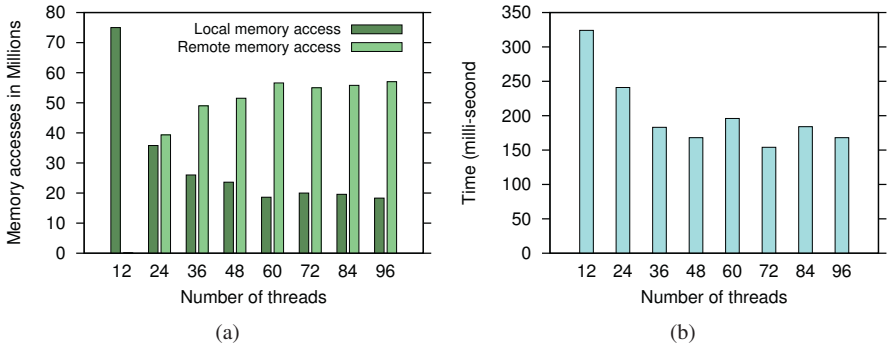

Figure 7.4: **Process and memory affinity to sockets controlled using numactl, for modified Q6. Buffer cache cleared.**

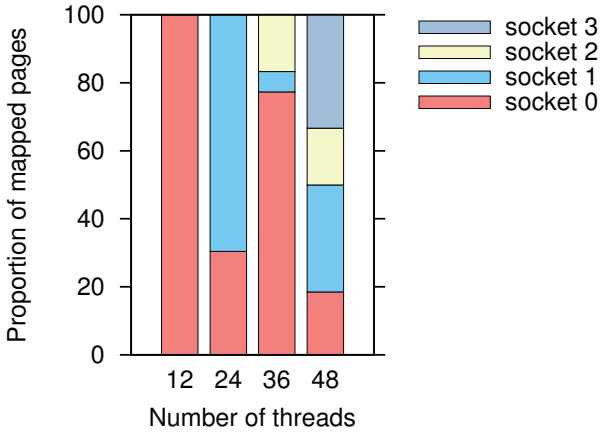

Figure 7.5: **Proportion of memory mapped pages on each socket when threads and memory allocation per socket is increased by including sockets one by one, using numactl, for modified Q6.**

/proc/process_id/numa_maps shows some of the remote accesses also arrive from the memory mapped libraries for the database server process.

We also expect some of the remote accesses to arrive due to cache coherency and thread migrations. As the number of threads increase, their migrations across sockets also increase, as shown in Figure 7.7. Numactl just prescribes the affinity across sockets, but at run-time the operating system is still free to do migrations to do load balancing [14]. For example, when process affinity is pinned to socket0 and socket1, operating system will not schedule threads on socket2 and socket3, however, it is free to migrate threads across socket0 and socket1, if the need arises. This explains why with an increase in the number of threads, the remote memory accesses increase, while the local memory accesses decrease.

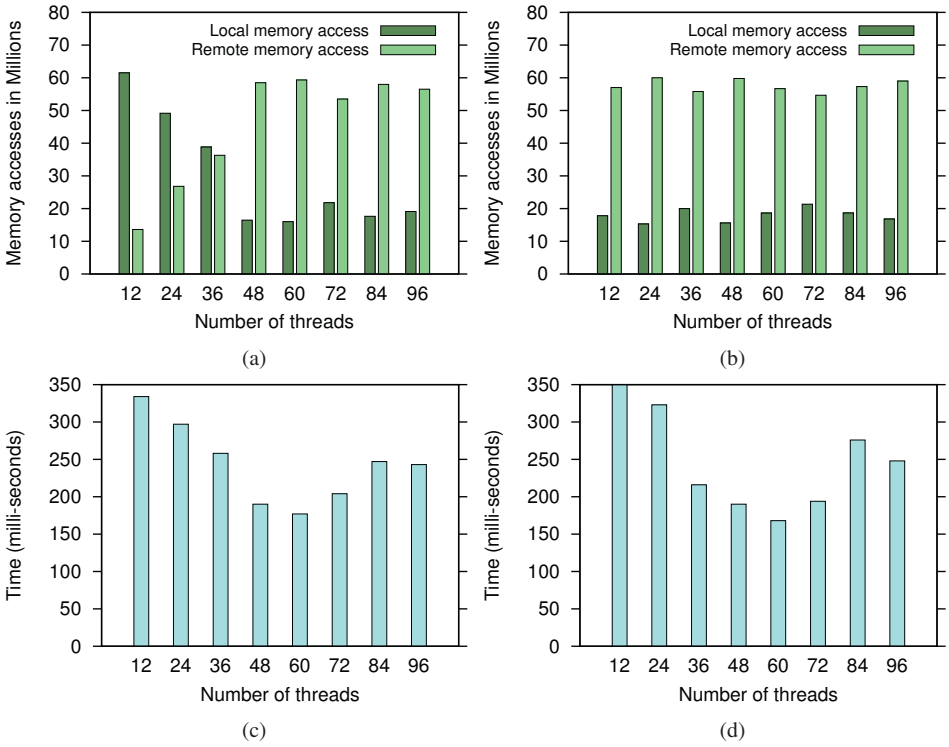

Figure 7.6: **Without process and memory affinity to sockets. a,c) Buffer cache cleared. b,d) Buffer cache not cleared.**

Execution without explicit affinity setting

Setup: Both Figures 7.6a and 7.6b show the number of local vs remote memory accesses, when the process or memory affinity is not set using numactl. The only difference being in 7.6a after each independent run, [3] the buffer caches are cleared using a kernel utility [4]. This is a crucial setting as if caches are not cleared memory mapped pages might stick around on previously allocated sockets, preventing their new locality based allocation. Figures 7.6a and 7.6b makes the difference prominently visible.

Figure 7.6a shows a pattern similar to Figure 7.4a, where the local memory accesses decrease with an increase in the number of threads. However, the change of both local and remote accesses for 12,24,36, and 48 threads in 7.6a is gradual, compared to a sudden change in Figure 7.4a. This indicates that when explicit affinity is not set, and when buffer caches are not polluted, the operating system does a good job of executing the process on sockets to maintain data locality. Both in 7.6a and 7.6b, the memory access pattern stabilizes when execution also uses hyper-threads starting from 60 threads, and almost matches the memory access

[3] An independent run occurs after 6 runs on the same configuration to take an average.
[4] echo 3 — sudo /usr/bin/tee /proc/sys/vm/drop_caches

pattern in Figure 7.4a.

Consider the case of 12 threads from Figure 7.6a, where the number of local accesses are less compared to Figure 7.4a. This is a result of the lack of process and memory affinity in Figure 7.6a execution. An important observation is unlike Figure 7.4a, the local memory accesses stay dominant than remote memory accesses until 36 threads, which indicates the operating system does an overall good job of scheduling. However, this does not get reflected accordingly in the execution times from Figures 7.4b and 7.6c, where for 24 and 36 threads, Figure 7.6a execution time should have been better than 7.4a. We are unable to explain this behavior.

Figure 7.6b offers an interesting perspective as well, as it shows without setting process and memory affinity to sockets, and without buffer cache cleared, both local and remote accesses almost stay constant irrespective of the number of threads in use. However, the execution time (See Figure 7.6d) does change and shows best time of around 150 sec, when 60 threads are in use. This seems very good as it indicates, without much efforts, just by choosing the correct number of threads, better execution can be obtained. However, finding the correct sweet spot in terms of the number of threads might not be that easy [3]. On the other hand, execution time for Figure 5a almost stabilizes after 48 threads are in use, which seems like a more robust approach.

Summary: Overall, we conclude that setting explicit process and memory affinity in NUMA oblivious plans, leads to a more robust execution as seen from Figure 7.4b, where the execution time stabilizes after 48 threads are in use. However, execution without process and memory affinity, without clearing buffer cache seems a more practical approach, and Figure 6d shows, it does offer similar execution time, but finding the exact number of threads to get the best execution can be tricky [3]. We also observe that the presence of hyper-threads has a negligible effect on the number of local and remote memory accesses.

Why remote memory access is bad? From Figure 7.8, the execution performance of modified Q6 in NUMA_Distr configuration is two times better than the NUMA_Obliv configuration. This indicates NUMA_Obliv shows relatively good performance overall. The loss of performance can be mainly attributed to the high number of remote memory accesses in NUMA_Obliv. This can be verified as follows. When the memory access is prominently local as in the case of 12 threads (See Figure 7.4a), the execution time is around 320 ms (See Figure 7.4b). If we divide the time by 4, since there are 4 sockets, the new time per socket is 80 ms, which matches with the NUMA_Distr execution of modified Q6 from Figure 7.8.

7.5 Pipe-lined execution comparison

Comparable performance is a subjective term. In our context we consider up-to 4 times difference as a comparable performance, whereas an order of magnitude improvement is considered worth the effort of a new system design.

Vectorwise (version 3.5) is a leading column store analytic system that uses pipelined vectorized execution. As it uses a dedicated buffer manager, rather than mem-

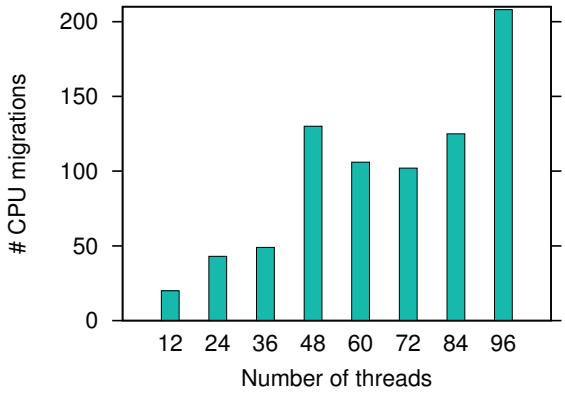

Figure 7.7: **Number of CPU migrations increase as the number of threads increase, for modified Q6.**

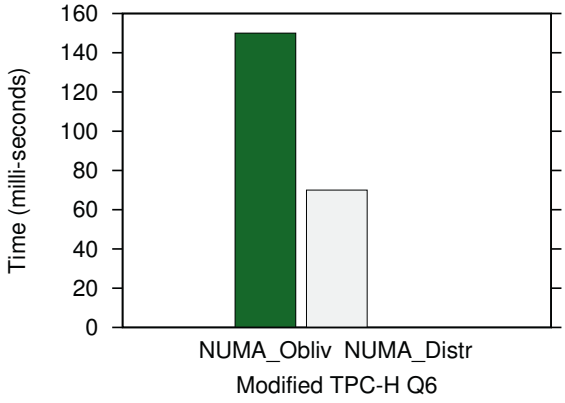

Figure 7.8: **More remote memory access in NUMA_Obliv slows down execution of modified Q6 by 2 times, compared to NUMA_Distr.**

ory mapped storage, a comparison with Vectorwise (See Figure 7.9) provides a perspective of the possible role of NUMA in its execution performance. The only configuration change we made is to enable histograms to generate better plans, and set parallelism level=96.

Vector_Def is the single instance default parallel execution without NUMA awareness and without affinity control. We compare it with MonetDB's NUMA_Shard configuration in Figure 7.9. Note that NUMA_Shard has just the lineitem and the orders table sharded and represented accordingly in the plan, however, the plan itself does not have any socket affinities as a single database instance is used. Hence NUMA_Shard configuration also represents NUMA oblivious execution (as under-

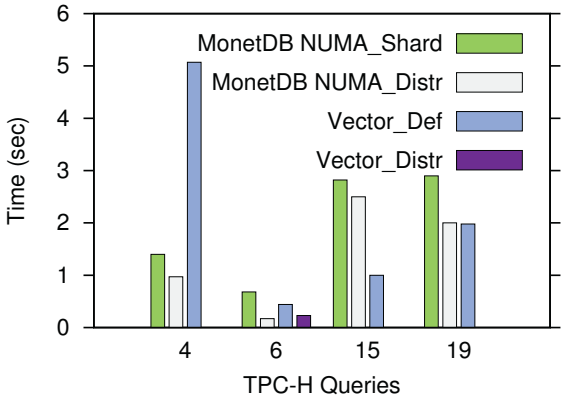

Figure 7.9: **Vectorwise's parallel execution compared to MonetDB's parallel execution (NUMA_Distr), for scale factor 100.**

lying storage is memory mapped), which we compare with Vector_Def configuration.

Vector_Def is relatively faster except for Q4, which hints at the executions with a NUMA oblivious buffer manager perform better than the executions with memory mapped buffers, as in MonetDB. We hypothesize that Vectorwise performs better even without NUMA awareness, due to its pipe-lined vectorized execution, and making it NUMA aware could improve its execution further. Depending on the query we observe a wide range of multi-core utilization (as reflected in CPU idleness using the *top* command), which we believe is a result of the cost model based parallel plan generation in Vectorwise. In [30] the authors illustrate problems of Vectorwise scalability beyond 8 cores due to locking and synchronization related overheads. We believe a NUMA aware approach similar to NUMA_Distr would benefit systems like Vectorwise to scale further, as it incurs minimal changes at the architectural level.

Vectorwise does not have a NUMA aware plan generation. As we do not have source code access to implement it, to get a perspective of the NUMA aware partitioned execution, we partition the lineitem table in four pieces. As query 6 is the simplest parallelizable query with select operations on the lineitem table, we measure its execution time on each of these pieces, and plot the highest time as Vector_Distr. We expect minimal aggregation overhead as the only aggregation operation is the sum of the four numbers from the four sockets, which is negligible compared to the individual select operation's time. Compared to Vector_Def, we observe an execution improvement of around 2 times.

Hyper's morsel driven parallelism is NUMA aware, where morsels from hash partitioned data are fed to the just in time compiled fused operator pipelines. From [104] Q6 takes 0.17 sec on 100 GB data-set on a 64 core (hyper-threaded) four socket, Intel Xeon X7560 @ 2.3GHz machine, with maximum QPI hop=1. While

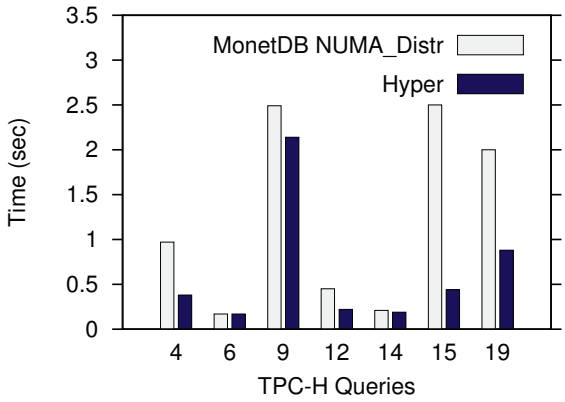

Figure 7.10: **Hyper's parallel execution compared to MonetDB's parallel execution (NUMA_Distr), for scale factor 100.**

Hyper's Q6 execution time is same as NUMA_Distr, its Q4 and Q19 execution is just 2 times faster, which can be due to less QPI traffic during join, due to hash based data partitioning in Hyper. Hyper is designed from scratch for optimal multi-core utilization and uses LLVM [103] generated just in time compiled fused operator pipelines. However, LLVM code generation also makes its code base much more complex.

In contrast, the promising query execution time for the queries 4, 6, and 19 by MonetDB's NUMA_Distr approach prompted us to explore more queries. We plot their execution time in Figure 7.10. It shows that the query execution performance of MonetDB's NUMA_Distr approach is comparable to Hyper's parallel execution performance for the query set under evaluation. In Figure 7.10 MonetDB uses 96 threads in total. To match Hyper's hardware configuration we restricted MonetDB's execution to 64 threads. Even with this change MonetDB's NUMA_Distr numbers do not show much variations compared to execution times in Figure 7.10.

Overall, considering its simplicity, NUMA_Distr approach looks promising for existing database architectures to control the problem of remote accesses, which results in a lower execution performance.

7.6 Related work

In [107] the authors evaluate the memory performance of NUMA machines. One of the main findings is how guaranteeing data locality to sockets need not be optimal always, due to increased pressure on local memory bandwidth. Authors provide use cases to show how a balance of remote and local memory accesses tend to balance out bandwidth for an optimal performance. Our calculations indicate for the Figure 7.4a, a local bandwidth of up-to 15 GB/sec for 12 threads, and a cumulative remote bandwidth of up-to 20GB/sec in 48 threads. In [111] authors also offer a detailed evaluation of memory performance in NUMA machines.

In [131] the authors treat the multi-socket system as a distributed system of individual database servers, in a master slave configuration. However, unlike our analytic workload, authors primarily explore the transactional workloads, from

throughput and client scalability perspective. Authors first elaborate how the traditional databases like MySQL and PostgreSQL fail to scale with NUMA systems and then propose a new middle-ware based system called Multimed that solves this problem, by using multiple database instances in a master-slave configuration. Data is replicated across all slaves, such that read only queries are handled by slaves, whereas the master handles update queries. Resource contention due to latching and synchronization in multi-cores is avoided by using multiple satellite database servers, instead of a single server. In our case by using multiple database servers affiliated with individual sockets, we try to minimize the remote memory accesses in analytical workloads, which we show is the prominent reason for the decreased query execution performance.

In [122] the authors treat sockets as hardware islands and explore the effect of different shared nothing database deployments from transactional workload perspective. The work is done in the context of SHORE-MT transactional system with a distributed transaction coordinator using two-phase commit protocol. Different possible deployment configurations are considered with different possible island formations, to explore its effects on the throughput of the transactional workloads by varying parameters such as the database size, granularity of partitions, skew, etc. This work is different from our work because we use analytical workloads which have different characteristics with long running queries, unlike transactional workloads where the queries are short and access a few rows only. We emphasize on improving the response time of individual queries by using query parallelization by range partitioning the data, while the transactional workloads prominently emphasis on the overall throughput. While the authors of [122] experiment with different possible database deployment sizes with different granularity of partitions, we use only two deployments, namely shared-everything (NUMA_Obliv & NUMA_Shard) and shared-nothing (NUMA_Distr), where we partition the large tables (lineitem & orders) in 4 partitions across the 4 sockets.

The NUMA architecture also influences new operating system designs. A new architecture, called multi-kernel [33] treats NUMA machine as a small distributed system of processes that communicate via message passing.

In [106] the authors show the case of NUMA aware algorithms, with a focus on data shuffling. A lot of work also focuses on NUMA aware operators, such as joins.

7.7 Summary

The research question: In this Chapter we address the questions, ”How well are the state-of-the-art database management system solutions using the available hardware resources?” and ”How to leverage multi-core systems to improve the performance of analytical workloads?”

NUMA systems pose a challenge to database engines as the memory access latency and bandwidth vary based on the location of the data access. Most server class systems use 4 socket NUMA systems. The approach taken to mitigate the NUMA problems is to build new database systems from scratch, making the in-

dividual database operators NUMA aware. However, most legacy database architectures can not afford to re-write their execution engines to make the operators NUMA aware, as its a massive engineering effort to re-write the legacy code. Making the legacy database systems perform optimally in a NUMA setting is thus a challenging problem.

The research contribution: How to affiliate data to memory banks to minimize cross socket data transfer is a fundamental problem in NUMA systems. The data affinity to the memory banks problem can be mitigated by letting the operating system do the scheduling using the memory mapping feature. Hence, we analyze the NUMA effects on a memory mapped storage system during parallelized query execution, and provide detailed insights into the observed behavior.

Based on these observations we propose a new shared nothing architecture, where the data is horizontally partitioned on each memory bank and each socket is affined with a database engine execution instance (slaves). A master database execution instance coordinates the distribution to slaves. This thesis proposes and analyzes the master-slave shared nothing architecture for the first time in the analytical database engine setup, while there is some related work in transactional database engines [131, 122]. Our proposed architecture is simple and does not require major architecture-level changes, hence, legacy database systems can make use of it to overcome NUMA related issues. The architecture brings down the cross-socket memory access interference, thereby improving the performance, which compares well with the latest state-of-the-art architecture, the Hyper database system [93]. We also show a micro-benchmark example of how Vectorwise [42], a leading database system can benefit from such an approach.

7.8 Conclusion

We analyzed the role of memory mapping in a NUMA system, by comparing NUMA oblivious vs NUMA aware plan execution in a database engine that uses memory mapped columnar storage. NUMA oblivious plans that execute using the operating system's default scheduling policy show relatively good performance. Remote memory accesses are identified as the main culprit in NUMA oblivious plans. When the database engine is used in a NUMA aware configuration by treating multi-socket CPU as a distributed system, remote memory accesses are minimal, leading to up-to 3 times improvement on the TPC-H queries tested. For the query set under evaluation, the distributed system based NUMA aware approach competes with the parallelism approach by the state-of-the-art systems such as Hyper.

Chapter 8

Database parallelism in many-core architectures

Query parallelization is traditionally done using CPU architectures that support multi-core, multi-socket, and GPU based architectures. Both multi-core and multi-socket CPUs have complex and powerful individual cores which are used either for intra-query or inter-query parallelization. GPU's offer a different level of parallelism where a portion of the query is accelerated using thousands of weak GPU cores. However, GPUs use a specialized programming paradigm which might be different from the traditional database engine programming paradigm.

The latest in the trend are the many-core CPUs from Intel also named Xeon-Phi, which offer a middle-ground between multi-core architectures and GPU architectures, by using X86 based programming over the GPU like architectures. In this Chapter we explore how to use the traditional database engine architectures to benefit from accelerated processing of many-core CPU architecture. The PCIe bus is a bottleneck when it comes to transfer of data between CPU and many-core CPUs. We compare different implementations to understand the effects of the PCIe bottleneck, and explore the plausible solutions to avoid them.

8.1 Motivation

The multi-core revolution is here. The number of cores keep on growing from tens of cores for single socket CPUs to hundreds of cores for multi-socket CPUs. The latest addition is the many-core CPU architecture by Intel, named as Xeon-Phi. This architecture acts as an accelerator in addition to the existing main processor, and is similar to a GPU based architecture. However, unlike GPUs, it uses a X86 based programming paradigm, making writing programs much easier as X86 programming is well established in terms of the availability of the tools, libraries, build environments, programming expertise, etc. One of the main application areas for the increasing number of cores is in the high performance computing (HPC) domain, however, other areas such as data analytics aligned fields also stand to benefit, and needs further exploration.

An interesting problem to explore is how to use Xeon-Phi's many cores to accelerate query processing in database systems. Query parallelization is a standard mechanism to improve analytic query execution. Query parallelization on traditional multi-socket multi-core CPUs face problems such as NUMA based access, finding an optimal degree of parallelism due to large number of cores, etc. The device memory on Xeon-Phi ranges between 6 GB to 16 GB depending on the model type, hence not all data can fit in the device memory. As Xeon-Phi acts as a co-processor on a PCIe bus alongside the normal host processor, one of the main problems is data transfer over the PCIe bus, so that the many cores of Xeon-Phi can work in parallel on this data. Limited device memory and constrained build environment due to cross compilation issues might also not allow execution of a complete database engine on Xeon-Phi. Hence, acceleration of specific database operators is of special interest.

In this Chapter we explore the possible ways to optimally do database operator accelerations on the Xeon-Phi co-processor by transferring data to the Xeon-Phi device memory, over the PCIe bus. We focus on the select operator's acceleration as our aim is to understand the data transfer properties over the PCIe bus. Offload processing is an in-built mode from Intel for Xeon-Phi based processing where the computation and data is automatically offloaded from Xeon based host over PCIe bus to the Xeon-Phi processor. In order to overcome the limitations of the offload mode, we implement a MPI based solution, and compare it with the offload mode based solution. We provide a detailed analysis of different types of data transfer over PCIe bus related experiments to get a better insight into related problems.

8.2 Contributions

PCIe bandwidth is a bottleneck during data transfer to Xeon-Phi. Low cost availability of Xeon-Phi prompts us to investigate if PCIe bottleneck could be mitigated such that instead of using a single Xeon-Phi, they could be used in a clustered configuration in a single system. Hence, we explore different possible implementations of data transfer over the PCIe bus to get further insights. Although our findings do not show performance improvement, we get some crucial insights into how the PCIe bottleneck behaves under different custom data transfer implementations.

The Chapter is structured as follows. In Section 8.3 we briefly describe the details of the Xeon-Phi architecture. Section 8.4 describes the offload mode of computation and the MPI based implementation. In Section 8.5 we provide the experimental results.

8.3 Xeon Phi architecture

The Xeon-Phi architecture code-name is Knights Corner. The co-processor is connected to an Intel processor also known as the "Host" through a PCI Express (PCIe) bus. Since the Xeon-Phi runs a Linux operating system, a virtualized TCP/IP stack can be implemented over the PCIe bus, which allows the co-processor to be used as a network node. Multiple co-processors can be installed in a single host system,

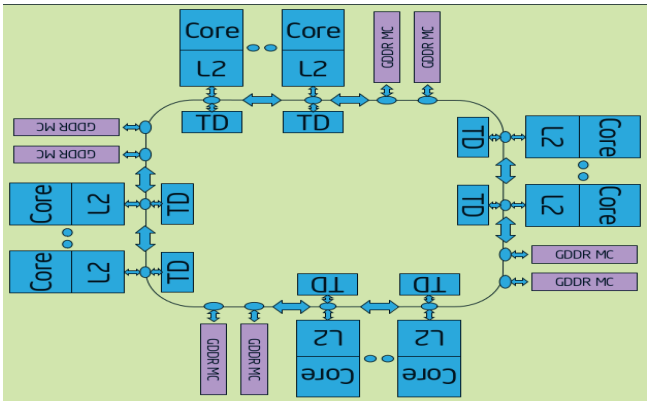

Figure 8.1: **Xeon-Phi with ring interconnect.**

and can communicate with each other over PCIe peer-to-peer interconnect without intervention from the host.

The individual cores in Xeon-Phi are based on the Pentium architecture, and are relatively much simpler than the complex powerful cores on the Xeon processors. One of the main reasons to use simple cores is to keep a low power profile. There are total 60 cores on Xeon-Phi and each core can use 4 hardware threads simultaneously, as shown in Figure 8.1, resulting in 240 hardware threads. It is recommended to run at least 2 threads per core as each core has 2 execution pipelines, and hence need at least 2 threads to utilize them fully. Xeon-Phi architecture is primarily composed of processing cores, caches, memory controllers, PCIe client logic, and a high bandwidth bidirectional ring-interconnect. Each core has a 32 KB instruction and data L1 cache, and 512 KB L2 cache that is kept fully coherent by a global distributed tag directory. The memory controllers and the PCIe client logic provide a direct interface to the GDDR5 memory on the co-processor and the PCIe bus. All the components are connected together by the ring interconnect.

An important unit is the vector processing unit (VPU), which features a 512 bit SIMD instruction set. Thus VPU can execute 16 single precision (SP) or 8 double precision (DP) operations per cycle. VPU also supports Fused Multiply Add (FMA) instruction and hence can execute 32 SP and 16 DP operations per cycle. It also provides support for integers.

The interconnect is implemented as a bi-directional ring. Each direction comprises of three independent rings. The largest is data ring, with 64 bytes width data to support high bandwidth transfer for cores. The address ring is smaller and is used to transfer read / write commands and memory addresses. The smallest ring is acknowledgment ring to send flow control and acknowledgment messages.

When a L2 cache miss occurs, the address request is sent on address ring to the tag directories. The memory addresses are uniformly distributed amongst the tag directories on the ring to provide a smooth traffic on the ring. If the requested data block is found in another core's L2 cache, a forwarding request is sent to that core's L2 over the address ring, and the request block is forwarded on the data block ring.

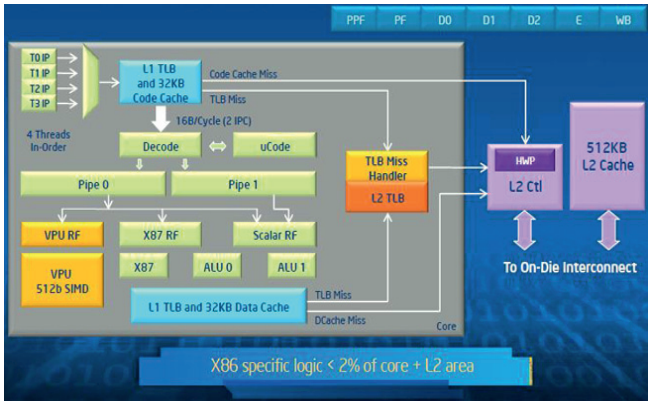

Figure 8.2: **Xeon-Phi core architecture.**

If the requested data is not found in any caches, a memory address is sent from the tag directory to the memory controller.

8.4 Data transfer over the PCIE bus

Xeon-Phi has a device memory that ranges between 6 to 16 GB based on the model type. The device memory is not sufficient to hold large data sets, and hence data needs to be transferred from the main memory of the host CPU to the device memory of the Xeon-Phi for doing any computation. PCIe bus however has a limited bandwidth and hence can not handle large data transfers.

We use two techniques, the Offload mode based and the MPI based, to transfer data between the host memory and the co-processor memory. We describe each of these techniques next.

8.4.1 Offload mode

Offloading is a technique by Intel to send the data and computation seamlessly to the Xeon Phi accelerator, from the Xeon based host, using the compiler's assistance. Offloading is done using pragma based directives as a part of the code generation. Intel provides an offloading supportive run-time environment which handles the necessary infrastructure. The offloaded code can be further parallelized using OpenMP based parallelization.

The offload pragma keyword specifies the clauses that contain information relevant to offloading to the target device. Target(mic:MIC_DEV) is the target clause for the compiler to generate code for both the host processor and the specified offload device. Offloading to multiple co-processors is possible using the correct target. The data to be copied to and from the Xeon-Phi is specified using In (var-list-modifier) and Out(var-list-modifier) clauses. The var-list-modifiers contains the information about the data to be transferred. Off-loading involves the overheads of marshaling of data, reserving and copying buffers on the host and co-processor

sides, and hence can be expensive as compared to a native execution. Native execution involves complete execution from the data stored in the device memory itself. Hence, off-loading is preferred if the overheads of transfer can be compensated by overlapping of data transfer with computations, and there is sufficient computational complexity. Some of the techniques used to avoid data transfer is to reuse the transferred data, and to reserve the buffers across multiple transfers. Next we provide a code snippet of the offloaded code.

```
_Pragma("offload target(mic) in(p,q,o,cnt,off,vl,
 src[p:q]) out(cnt,dst[0:BATcap])")
{
  _Pragma("omp parallel num_threads(120)")
  {
     _Pragma("omp for reduction(+:cnt)")
        for (;p < q; p++)
        {
             o = (p+off);
             v = src[o-off];
             dst[cnt] = o;
             cnt += (v>=vl);
        }
   }
}
```

The offload mode of execution handles the data transfer seamlessly using Intel's offload run-time environment, thereby making it difficult to have a control over the data transfer. In-order to be able to get more insights into the data transfer, we device our own implementation of data transfer using the message passing interface (MPI). We describe about it next.

8.4.2 MPI Mode

Message passing interface (MPI) is a language independent distributed message communication protocol used in a distributed system setting. MPI uses explicit data transfer routines to transfer data between multiple processes (also known as ranks) on the same / multiple nodes in a cluster. MPI's goals are high performance, scalability, and portability.

There are multiple MPI implementations in use, such as OpenMPI, MPICH, MVAPICH, and Intel MPI, being prominent ones. The implementations consists of specific set of routines directly callable from C,C++,Fortran and any other language able to interface with its libraries, including C#, Java, Python. MPI_Send and MPI_Receive APIs are used to transfer data between different processes. Both of these APIs use blocking or non-blocking mode of data transfer. We use the open-source MPICH3.1 implementation due to its support for multi-threading and profiling in Xeon-Phi environment.

We devise our own implementation of data transfer between the host memory and the Xeon-Phi device memory using MPI. We categorize it into a single buffer transfer mode and a streamed buffer transfer mode. In a single buffer transfer mode, the computation on Xeon Phi has to wait until all the data is available in the device memory. It also does not utilize the bi-directional PCIe bandwidth optimally, as the PCIe bus is busy in the data transfer only in a single direction. These problems are avoided in the streamed buffer transfer mode, where the data is split into multiple vectors, and each vector is sent one by one. When the first vector arrives at Xeon-Phi, the computation could start using it, while the next streamed vectors are being transferred on the PCIe bus concurrently.

8.4.3 MPI profiling

Being a distributed message passing protocol, MPI is highly susceptible to performance problems due to optimal tuning. Some of the prominent tuning parameters include the message size, the buffer size, the collective algorithms, the underlying protocols to use. MPI offers a profiling interface as a part of its standard distribution, which can be used to profile applications making MPI calls. This interface is called PMPI. The PMPI interface calls match the standard MPI calls but with a different name and act as a wrapper to set the relevant tuning parameters around the regular MPI calls.

Though PMPI interface is standard on all MPI implementations, it does not provide details on the internals of the MPI library performance. Another new interface which provides these details is MPI_T, which is based on profiling variables. This interface allows to probe for two types of variables, control and performance variables. The control variables are used for controlling the MPI performance such as the ”eager limit” threshold which controls whether the eager protocol or rendezeneous protocol is chosen, based on the size of the message buffer. The performance variables are used for probing the MPI library internal details such as the message size, buffer size, etc.

Next we describe the architecture of our implementation for MPI based data transfer over the PCIe bus.

8.5 MPI data transfer Architecture

Due to the constraints on the Xeon-Phi device memory size and possible cross compilation issues during the build environment, having a full fledged database execution engine executing on Xeon-Phi is not practical as of yet.

Hence, we choose the model of operator based acceleration where a few operators are accelerated on Xeon-Phi. For example, in the current implementation we use a selection operator acceleration, where the input data to the select operator is transferred to the Xeon-Phi device memory from the host memory. The selection operator’s computation is accelerated on Xeon-Phi and the generated output is shipped back to the host memory.

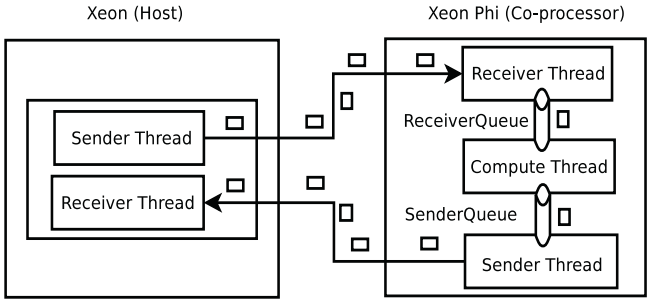

Figure 8.3: **Multi-threaded architecture.**

We use MPI based data transfer between the host memory and Xeon-Phi device memory, where we use a single host and a single Xeon-Phi accelerator. Another possible variation is single host and multiple Xeon-Phi accelerators.

8.5.1 Single host- Single Phi acceleration

Figure 8.3 shows the example of single host - single phi based acceleration. Here the Xeon based host uses a multi-threaded implementation with a separate *sender* and *receiver* thread to transfer and receive the streamed data to and from the Xeon-Phi co-processor. The Xeon-Phi also uses a multi-threaded receiver and sender to receive and send the data from and to the Xeon based host. The computation on the received data is done in a separate thread. Two separate queues are used to gather the data received from the host and to be sent to the host after computation on it is done. The receiver thread on Xeon-phi receives streamed data, and puts in a receiver queue. The compute thread picks up the streamed data from the receiver queue as it becomes available and does the computation to generate the output streamed data, which it puts in the sender queue. The sender thread picks up the data from the sender queue and sends it to the Xeon host.

8.6 Experiments

We use MonetDB, the open-source column store database for our evaluation. Our experimental platform consists of a Xeon based host with Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz with two sockets with 8 cores each with hyper-threading enabled. 256 GB of DDR3 memory and Enterprise Linux 7 Operating System installation. The Xeon-Phi model being used is 5110P with 8GB of device memory. All experiments use 120 threads of execution on Xeon-Phi.

During the experiments we do a comparison of offload mode of data transfer against our MPI based data transfer over the PCIe bus for select operator acceleration using different dataset sizes.

We use TPC-H benchmark data sets of varying sizes to get a perspective of the effect of the data set size. We use a single simple query to analyze different cases.

Select count(*) from part where p_size >5;

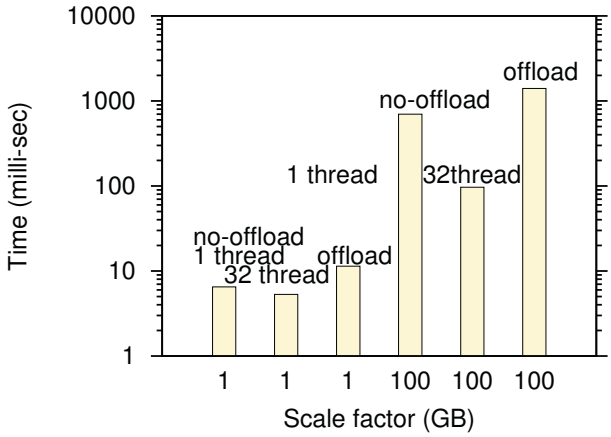

Figure 8.4: **Single thread vs 32 threaded vs offloaded execution.**

8.6.1 Offload-data transfer based execution

In this experiment we measure the query execution time using offload mode of data transfer. During offload mode of data transfer only the loop that does selection on the columnar data is offloaded automatically using compiler generated code.

Graph 8.4 shows the execution time for a single threaded execution, 32 threaded execution on Xeon host, and offloaded execution with 1 thread on Xeon based host and 120 threads on Xeon-Phi. Please note that the Y-axis is log scale.

For 1 GB scale factor when the execution on Xeon based host goes from serial execution to 32 threaded parallelized execution, timing improves but not as much as it improves for 100 GB scale factor. The Offloaded execution uses a single thread on Xeon based host side, whereas the Xeon-Phi uses 120 threads to parallelize the selection based for loop code using OpenMP pragmas. We can observe that the offloaded execution does not improve the execution time much compared to serial execution on Xeon. Most of the time is spent in data transfer over PCIe as seen from Table 8.1.

Table 8.1: **Offload execution time split-up**

	Computation	Data transfer
1 GB	1 ms	8.5 ms
100 GB	40 ms	1110 ms

During offloaded execution a single buffer gets transferred over the PCIe bus to the Xeon-Phi. To understand the effect of a streamed execution we vectorized the single buffer and vectors are passed over the PCIe bus so that execution can start on individual vectors. The vectorized implementation uses MPI to do data transfer, which we explain next.

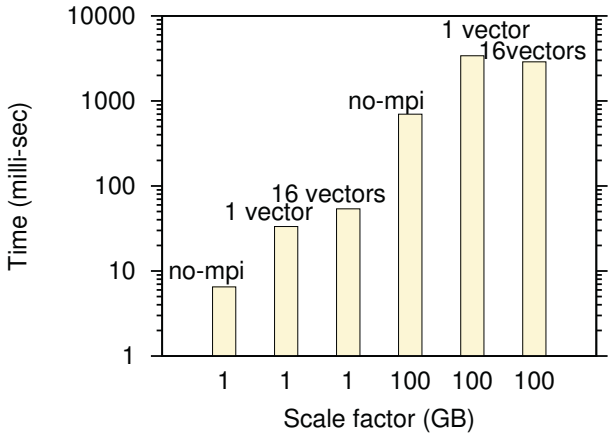

Figure 8.5: **Single thread single vector (no-mpi) vs MPI based streaming execution.**

8.6.2 MPI based data transfer with streaming execution

Graph 8.5 shows the execution time of a non-MPI single vector based execution compared with a MPI based streaming execution with a single vector and 16 vectors. Single vector indicates lack of vectorization where the entire buffer is transferred over PCIe as a single vector compared to when the buffer is vectorized into 16 vectors.

In both 1GB and 100 GB scale factor case no-mpi based execution where execution happens over a serial plan performs better than mpi based executions. Digging into literature indicates MPI has too much overhead when the TCP stack is used to do data transfer due to kernel context switches and buffer copying.

To investigate if the TCP stack overhead could be mitigated one possible solution is to use a different fabric using Infiniband stack for communication over RDMA. Our efforts to configure an Infiniband stack fabric for Xeon-Phi were not successful due to heavy dependencies on SDK versions, kernel versions, and constant ongoing development to improve the fabric drivers. However, this remains a future investigation possibility.

8.6.3 MPI tuning

Our experiments to analyze the effect of tuning of different MPI tuning parameters on the MPI performance did not result in significant differences. We were specifically interested in the parameters that include the message size, the buffer size, the collective algorithms, the underlying protocols to use.

We experimented with ”eager limit” threshold which controls whether the eager protocol or rendezeneous protocol is chosen, based on the size of the message buffer. However, we did not observe a significant difference in execution times.

For 100 GB data-set, the size of the individual vector (vector_size) passed on PCIe was 5000004 bytes. The eager protocol gets used when the tuning parameter MPIR_CVAR_CH3_EAGER_MAX_MSG_SIZE >vector_size (MPIR_CVAR_CH3_EAGER_MAX_MSG_SIZE=6000004). The total execution time in micro-seconds is 1137231.

The rendezeneous protocol gets used when the tuning parameter MPIR_CVAR_CH3_EAGER_MAX_MSG_SIZE <vector_size (MPIR_CVAR_CH3_EAGER_MAX_MSG_SIZE=1000004). The total execution time in micro-seconds is 1001207.

The execution time difference of 0.1 seconds is a result of the ”eager limit” threshold tuning.

8.7 Summary

The research question: Intel Xeon-Phi is a many-integrated-core (MIC) architecture primarily targeted towards high performance computing (HPC) workloads. Exploration of Xeon-Phi as a co-processor to accelerate database query execution is an active research area [89, 47, 145]. First generation Xeon-Phis (Knights Corner architecture) have a limited device memory (up-to 16 GB) and work as a co-processor attached to the PCIe bus, which has severe constraints on the data bandwidth (up to 6GB/sec practically observed when compared with the memory bandwidth). It limits their usage in data heavy workloads. Hence, optimizing PCIe bandwidth usage is an important problem in the context of database workloads.

The research contribution: We propose a many-core architecture execution engine for database query acceleration. We investigate the PCIe bottleneck for the data transfer to Xeon-Phi. Our proposed execution engine uses a streaming based multi-threaded MPI implementation to accelerate a *select* operator on a Xeon-Phi. As Xeon-Phi prices have dropped from thousand dollars to a hundred dollar, more than one Xeon-Phi could be used in a cluster configuration in a single system for acceleration. Our MPI based implementation is an exploration attempt in that direction.
At the time of writing this thesis a new advanced architecture, Knights Landing, has been launched in the market, and it takes care of the PCIe bottleneck as the new architecture can work as a standalone processor, instead of only as a co-processor on the PCIe bus. The hardware landscape thus evolves rapidly. However, the cheap prices of the old architecture compared to the new architecture still makes it attractive to use them in a clustered configuration in a single system.

8.8 Conclusion

Intel’s many-core architecture offers large number of cores at low energy footprint with 512 bit SIMD registers. Xeon-phi still has a long road to go to become useful in the database world where large sized data-sets are often a norm. Xeon-phi is useful for compute heavy functions that could be possibly accelerated such as spe-

cial statistical functions in *python, R* etc. inside databases. The main hurdle is data transfer over the PCIe bus since the device memory of Xeon-Phi still remains relatively small in size to fit database analytical data-sets. Our efforts to understand the effect of data transfer using offloaded mode of execution and a MPI based streaming based execution did not provide breakthrough results in terms of performance improvement, however, it provided us with detailed understanding of the MPI based performance issues and possible ways to resolve them, which should help in any future investigations of related topics. Intel's knights-landing architecture which is due in 2016, promises to remove many bottlenecks such as the device memory size bottleneck by providing a much larger device memory size and a standalone processor. It hopefully will help database analytical workloads to make better use of these many-core architectures.

Chapter 9

Summary and future directions

The multi-core CPU landscape is vast. Most database researchers exploring multi-core related research focus on individual operators and study their behavior, propose new algorithm implementations, and analyze the speedup of comparable algorithms using different number of cores and CPU architectures. Most of the research literature comprises of such papers.

However, often this research lacks a perspective to look at the problem of database query parallelization in a holistic manner, which involves taking into account the combined effects of system execution of an entire database execution engine. Query parallelization is a very hard problem and most systems struggle to get it correct due to different factors involved such as software complexity, hardware complexity, CPU architecture dependency, etc. For example, a join operator's performance is different in an isolated setting in a standalone implementation, compared to when executed in a full-fledged execution engine. The performance gets affected due to run-time resource contention, changing workloads, type of queries under execution, different operators degree of parallelism, etc.

This thesis focuses on the entire ecosystem of query parallelization and related problems in the context of a well established columnar database system. It proposes techniques for improvement and provides detailed experimentation comparing with state-of-the-art columnar systems such as Hyper and Vectorwise. Columnar database systems are preferred for studying analytical query execution unlike row-based database systems, as columnar database systems are designed and optimized from grounds up for analytical query processing. The research in this field hence comprises of creating an ecosystem of different aspects such as 1) new tools to identify performance bottlenecks in execution engines and propose solutions, 2) proposing new plan parallelization techniques based on the observed behavior and validating the proposed techniques through detailed experimentation, 3) analyzing the effects of these techniques on the latest hardware, and 4) exploring the latest hardware characteristics to understand their effect on query execution improvement.

Gaining access to a full-fledged columnar database system's source code and source code related expertise is difficult. As a result, no PhD thesis in the database

systems literature explores the analytical query parallelization problem in a holistic manner the way this thesis explores it. Making a complex piece of software like a database execution engine work perfectly is very difficult and any holistic explorations provide new insights and techniques into its improvement. The research in this thesis contributes towards it. Wherever possible the experiments use state-of-the-art systems such as Vectorwise [42] and Hyper [93] for comparison and provide suggestions on how these systems could be improved further. Database architecture systems research is mostly applied systems research, and this thesis takes the applied approach, as modeling an entire database execution engine is known to be in-feasible.

The first part of this Chapter summarizes the research explorations undertaken in this thesis, providing the highlights of the scientific contributions of each Chapter.

The second part of the Chapter provides a brief overview of the upcoming new trends in the hardware, and the role of database system integration and related research problems.

9.1 Chapter 3 & 4

If I have seen further it is by standing on the shoulders of giants - Isaac Newton

Research in applied sciences such as Computer Science and Database Systems is mostly engineering driven. Database systems are a complex piece of software, and tools that assist in getting insights into their behavior is a crucial part of the research ecosystem. Chapters 3 & 4 describe two such tools *Stethoscope* and *Tomograph*, that we implemented to visualize execution of the parallel query plans. Parallel plans quickly become too complex for performance troubleshooting analysis. Most database systems still rely on text based tools, which are good for analyzing serial query plans, but are poor for parallel plan analysis due to their complexity. As the number of cores increase, effective analysis of these plans is a crucial step in improving the performance of the parallel plan execution. We analyze different use cases of performance bottlenecks and provide possible solutions for improvement. Hence, visualization of query execution plans acts as stepping stone in the parallel query execution research ecosystem. The techniques we propose in the rest of the thesis derive from our observations of different plans analyzed using these tools.

Our work has inspired similar tools in commercial systems such as Vectorwise, HP Vertica, and SAP Hana.

9.2 Chapter 5

Query execution engines are a complex beast. Query parallelization is a well known technique to make query execution faster. The degree of parallelism of operators in the query execution plan is the most influential factor in query execution. Hence, getting it correct is crucial. Most systems use a cost model or a heuristic based approach to determine the degree of parallelism of a query plan. We propose a

new technique that is adaptive and feedback-based called *adaptive parallelization*. When the same query pattern is fired multiple times, adaptive parallelization comes into effect. It involves incrementing the degree of parallelism of the most expensive operator with each execution feedback.

The process of adaptive parallelization should also converge. We propose a new convergence algorithm that stops the feedback loop when the optimal degree of parallelism for the query plan has been determined, and a global minimum execution time has been reached. Detailed experimentation shows that the adaptive plans perform comparably to heuristic plans under isolated execution. Adaptive plans also exhibit much better multi-core utilization which helps during concurrent workload execution. The convergence algorithm shows consistency in observed convergence runs, hence behaves robustly.

Though adaptive parallelization seems like a relatively simple technique, getting it to work correctly in a complex column store architecture like MonetDB is a major challenge due to the complexity of MonetDB plans. Plan modification based on feedback quickly becomes very complex due to operators interdependence. We modify implementations of all the fundamental relational algebra operators and some auxiliary operators that are column store specific to make them adaptively parallelizable aware.

We provide details about how adaptive parallelization could be used by other *Exchange* operator-based database systems. Crucial insights are provided describing the effects of correct number of data partitions on the multi-core utilization, resulting in less resource contention for adaptive plans compared to heuristic plans. The convergence algorithm we propose is relatively simple and should be easy to adapt to other systems. Adaptive parallelization has generated good interest into research community as observed during conference discussions and follow up research undertaken at other research centers.

9.3 Chapter 6

Query parallelization is challenging because determining the optimal degree of parallelism is a difficult problem. It also gets affected drastically due to run-time resource variations during concurrent workload. Almost all database systems generate a query plan without taking into account the concurrent workload, as modeling a concurrent workload during query plan generation is really complex. The query plan generated in this manner thus behaves sub-optimally under concurrent workload due to run-time resource constraints put by concurrent queries under execution.

Getting insights into how a parallelized query plan behaves under concurrent workload is thus very crucial and the research exploring this problem is very limited. In Chapter 6 we compare three different database parallelization techniques (cost model, heuristic, and adaptive parallelization) using three different database systems to gain insights into how a single parallelized query gets affected by concurrent queries under execution. A key finding is that the total number of data partitions in adaptive plans being less put less resource constraints on the resources

such as the CPU cores and the memory, allowing adaptive plans to perform better. We provide detailed insights into hardware characteristic effects at the architecture level by quantifying L1 / L3 cache misses, pipeline stalls, etc. to show the effect of different workloads on performance characteristics of query execution. We show that inter query parallelization as used by legacy systems such as Postgres perform very poorly compared to intra-query parallelization by systems such as MonetDB and Vectorwise. However, Postgres shows robust behavior compared to these other systems. Overall, adaptive plans perform better. We show that using random workload leads to less resource contention compared to workload with similar queries, when the database systems do not employ data sharing techniques. This is a crucial insight when it comes to mixing of different types of workloads during concurrent executions. No other research has studied this problem in such a depth from an analytical query execution perspective so far. Hence, this Chapter provides crucial scientific contributions in understanding how concurrent workloads affect parallelized query execution.

9.4 Chapter 7

As addition of more cores to an existing CPU socket has reached threshold, new architectures based on multi-socket systems are becoming a norm in server class systems. These systems are termed non-uniform memory access (NUMA) systems, as each socket's data access time varies based on the memory bank being accessed. Query execution on multi-socket systems quickly runs into problem as data access latency and bandwidth across sockets vary depending on the memory being accessed.

We explore if the memory mapping feature by the operating system can resolve these issues, as operating system takes care of mapping data affinity to different memory banks of different sockets. We perform a thorough analysis of the resulting access patterns to get insights. Based on this analysis we propose a new distributed shared nothing architecture based on master-slave configuration. The multi-socket system is treated as a shared nothing system by horizontally partitioning data across different memory banks and restricting one database execution instance on each socket, in coordination with a master node. We show that this master-slave architecture helps with restricting cross socket memory accesses. This results in improved query execution performance comparable to the latest state-of-the-art system, Hyper. We elaborate how this simple scheme could turn any existing database system into a distributed database system in a multi-socket environment. We provide a sample use case with the state-of-the-art system Vectorwise. This is the only research work in the context of analytical database systems that introduces a shared nothing architecture, while similar work exists for transactional database systems.

9.5 Chapter 8

In this Chapter we explore how columnar database systems can exploit a new CPU architecture termed ”Intel Many-Core architecture” pioneered by Xeon-Phi family of co-processor CPUs. Xeon-Phi is primarily targeted at HPC workloads, however, how to utilize their large number of low power cores in database workloads is an interesting question to investigate. Xeon-Phi acts as a co-processor connected to the main processor through the PCIe bus. PCIe bus is well known for its bandwidth bottleneck problem where practical bandwidth of up to 6GB/sec could be reached using standard benchmarks.

We investigate how can we further optimize the use of the PCIe bus for data transfer for an operator's acceleration in a database execution engine, by using MPI based vectorized multi-threaded streamed execution. We compare this with the offload based execution provided by default by Intel. The PCIe bottleneck is well known for data transfer, however, since Xeon-Phi prices have dropped from thousands to a mere hundred dollars, they could possibly be used in a clustered configuration in a single system. The MPI based streamed execution engine is a step in that direction. Our experiments using MPI run into issues of MPI overhead due to TCP stack overheads, which we had not anticipated before. Similarly setting up an Infiniband based setup also ran into issues due to version dependency on kernel and related problems. This is the first of its kind of study that implements an MPI based streamed vectorized execution engine and hence provides crucial insights into what could go wrong with such a system. Though results do not show improvements, it shows how bleeding edge technology requires time before it could be fully exploited in a database system.

The author's internship experience at Oracle Labs in Silicon Valley on a new software-hardware co-design based columnar multi-core system engine is also valuable in this context. For a couple of years Oracle Labs is designing a low power multi-core hardware chip to be used in distributed clustered setting with a columnar database accelerator execution engine. It also acts as a proof that new hardware takes a long time to mature, before it could be optimally used by the software.

Having done an overview of the scientific contributions of the thesis, next we provide a brief overview of how hardware is evolving and what role the database systems have to play in this evolution.

9.6 Knights landing- Many core architecture

Knights Landing is the next generation architecture after Knights Corner architecture in Intel's Xeon Phi family of many-core processors. Along with 72 Atom based cores (4 threads per core), it also has a 16 GB of on-board high speed memory (MCDRAM), which Intel claims to be 5 times more power efficient than GDDR5 and 3 times more dense. There are 6 memory channels of 384GB DDR4 memory. It also has 36 PCIe lanes which can host 2 Knight's Corner architecture cards. The large device memory allows to fit the big data sets in the device memory itself for the native execution, thereby getting rid of the need for the data transfer over the PCIe

bus.

Another major upgrade is, Xeon Phi can now act as a solo processor in its own socket, with a fully functional operating system. It can also continue to act as a co-processor over the PCIe bus. A new interconnect called Omni-connect is also introduced with 100Gbps bandwidth and is supposedly faster than the Infiniband interconnect.

The main problem faced by database systems while using the Knights corner architecture is the lack of sufficient device memory to fit the big data-sets. This requires copying of data over the PCIe bus. The presence of 348 GB of device memory solves this problems to a larger extent as many data-sets can now fit in the device memory itself.

Since Xeon-Phi is a processor with many-core architecture, the database systems software needs to be cross-compiled to be able to execute on it. Depending on the complexity of the build requirements dependency, cross-compilation could get tricky, and in turn hamper the attempts to build the database system for Xeon-Phi.

The presence of 4 threads per core on a 60 core Xeon-Phi allows the total number of threads to be 240. How to effectively parallelize the query plans to utilize so much compute power is a challenge. Xeon Phi has a new type of memory called on-chip MCDRAM memory (16GB). The presence of this memory changes the traditional cache based memory access patterns considerably, as this memory now acts like a L3 cache. NUMA aspects of data access also have to be considered.

Hence, the Knights Landing architecture brings new aspects to the database system execution engine design and relevant research needs to be done to make optimal utilization of this HPC targeted many-core CPU for database workloads. The results described in Chapter 8 can provide a reference in this direction.

9.7 Internet of things hardware

The computer industry has gradually transitioned from PCs, laptops, smart phones, and tablets to sensor based ubiquitous devices. Internet of things is the name for the sensor based technology ecosystem trend under development. The devices that fall under Internet of Things include fitness wearable devices, home automation devices, retail sensor analytical devices, etc. The estimation is that there will be around 50 billion objects forming part of the Internet of Things infrastructure by 2020.

As the individually connected objects grow, they would also need distributed hubs with processing power, that could act as miniature versions of the powerful servers. We can already see a trend for this miniature server type of systems, in the form of system on chips and small form factor single board computers. These would be the processing nodes for the ubiquitous devices of the Internet of Things infrastructure. Examples of some single board computers include Raspberry Pi, Intel Galilio, Intel Edison, Cubieboard, Parallella, Arduino and other system on chips. Most of these boards have sufficient power in their small form factor, with at least two processing cores, 1GB RAM, 2 GB flash ROM, etc. For example, Raspberry Pi 2 model B has a quad core 900MHz ARM based CPU and a VideoCore

GPU. Intel Edison has a dual core Intel Atom 500 MHz CPU and a quark processor, 1GB RAM, 4GB flash, onboard wifi and blacktooth, and SD card sized form factor, which is extremely crucial for embedded devices.

The Internet of Things objects are going to generate a constant stream of data. This data needs to be stored, processed, and analyzed to derive meaningful analytics out of it. The single board systems we saw earlier are going to play a crucial role in this development where they could form a middle layer before the data is finally stored on the back-end infrastructure. These single board computers already have parallel hardware in terms of multiple cores, and the number of cores are going to increase substantially as can be seen already by the trend in Raspberry Pi where there are now 4 cores compared to 2 cores when it was introduced 2 years earlier. Another board named Parallella has two ARM based cores, whereas 16 cores form its on-board co-processor. As these computer boards start getting more number of cores, they will give rise to new challenges in parallel processing.

However, unlike the desktop or server based systems these boards would have constraints on their power usage, device memory, persistent storage, etc. and the softwares designers need to take that into account. Since one of the main functions of these boards would be to act as data aggregators before sending the data to higher level infrastructure, the role of database systems become crucial from data processing perspective. Hence, their design needs to be thought from new hardware requirement perspective. The embedded database systems and stream based database systems might be able to fit the requirements of Internet of Things based ecosystem. Already attempts are being made to tune the existing designs or develop new database engines from scratch to suit the streaming nature of data in Internet of Things infrastructure.

This opens up new research directions to design the database system components such as the query language, query optimizers, cost models for new hardware, buffer management schemes, query execution engine paradigms, query plan parallelization, etc. for the new class of hardware devices with all their new constraints. Big players such as Google and Intel are already working towards providing a platform based architecture to provide a standardization for Internet of Things ecosystem. Google announced the Project Brillo in October 2015, which aims to create an Android based operating system infrastructure platform for Internet of Things devices. Intel is already working towards making its processor ecosystem fit this new platform. Database systems will also have to be a part of such drive to bring a standardized ecosystem for better inter-operabilitiy between different devices generated data to draw meaningful analytics out of it.

One of the challenges is also going to be creating diverse build environments for such a diverse set of hardware, as more and more players start building their own hardware. The role of community based ecosystem matters the most to help such diverse hardware take off the ground and succeed in the longer run. For example, Raspberry Pi is a success story as it was one of the first that came into the market, and now has a full-fledged community backing it. As more and more new hardware boards emerge this is going to be a challenge and only the best will survive. Hence, software design is going to be challenge in such a competitive dynamic ecosystem.

9.8 Mobile processors

Many high end mobile devices already are powered by 8 big.LITTLE ARM architecture based cores, about which we provided a brief overview in the second Chapter. The trend of increasing number of cores is going to continue with mobile device processors too.

ARM architecture based processors dominate mobile processor market, some examples being Samsung Exynos, Qualcomm Snapdragon, Nvidia Tegra and Apple A7 platforms. Intel is trying to expand into this market with its own Atom series of mobile processors. The main concern in mobile processors is to conserve the power. ARM with its decades of experience in mobile processors has simple RISC based architecture compared to Intel Atom processors CISC architecture, which look more like stripped down version of the desktop and server processors where Intel dominates. Hence, mobile based processors are going to see a lot of innovation happening to be able to overcome the limitations of existing mobile based processors.

Mobile devices already surpass the desktop and laptop computers in the world. Smart-phones of the future are going to resemble desktop PCs in their computational power. With cloud based storage mobile app based ecosystem will advance further in terms of the kind of applications possibly hosted on mobile based devices. Database systems would play a crucial role in such an environment. Hence, their role needs to be re-thought carefully in this new mobile based environment, as all the database systems are primarily designed for desktop and server level systems. There are many potential research opportunities in designing new power aware database systems for the mobile based systems, where traditional database systems need not be optimal both from their power usage and performance perspective, given their dependence on the desktop and server level hardware architecture.

Screen form factor of mobile based devices also would play a crucial role in how the user interfaces for database systems on mobile devices look like. There are already attempts to build touch screen based database system query interfaces. More such visualization based interfaces would arrives soon that would provide ease of interaction with the data, with the mobile devices acting as an agent providing that access.

9.9 Summary

As the number of CPU cores increase, making an optimal use of them during analytical query parallelization is a critical problem. First half of this Chapter provides a summary of the research questions addressed and the research contributions made in each of the thesis Chapters, in the context of analytical parallelized query execution. We provide a brief summary next.

Most database systems use either a heuristic or a cost model based parallel plan generation approach which does not generate efficient parallel plans. In this thesis we propose a new parallel plan generation technique called *adaptive parallelization* and explore it in the context of different CPU architectures. We propose new

visualization tools to identify query performance bottleneck issues. We show effectiveness of these tools in identifying and resolving the performance bottlenecks. Next, we investigate the critical problem of the effect of resource contention due to concurrent workload on a parallelized query execution in the context of three different state-of-the-art database systems. We continue our exploration to multi-socket systems and propose a new shared nothing system architecture for legacy database systems, to mitigate the NUMA related remote memory access problem. Towards the end we investigate the role of PCIe bandwidth bottleneck during data transfer in Xeon-Phi many-core architecture, in the context of database workloads.

In the second half of the Chapter, we give an overview of the emerging trends in the new hardware and what challenges they pose to the database systems from the design and the architecture perspective. We give a brief overview of how the multi-core hardware would possibly evolve with respect to different CPU architectures, and the design choices for their corresponding applications with data management as a focus. As the data continues to grow rapidly, the role of database systems assume a lot of importance in the new data driven ecosystem. Making the database systems use the emerging hardware technologies optimally is a challenging task, and holds a lot of research potential. The extensive experimentation provided in this thesis can serve as a reference and guide to explore these new systems.

9.10 Conclusion

Query parallelization is an important research problem as it directly affects the query execution performance. The focus is to improve the query execution response time of long running analytical queries on different multi-core CPU architectures. As column-store database systems are designed to provide optimized performance for analytical query execution, getting them to perform optimally with the ubiquitous present multi-core CPU architectures is also critically important.

Multi-core CPU architectures vary a lot ranging from single socket systems to multi-socket systems to low power many-core systems. Making columnar database systems perform optimally with these different CPU architectures requires a thorough analysis of different software and hardware components, with a software-hardware co-design oriented approach. The software complexity and the restrictions on the access to the source code makes it difficult to do a holistic evaluation of the query parallelization problem with different multi-core CPU architectures. Hence, the holistic exploration of the query parallelization problem in the context of a full fledged open-source columnar database system, *MonetDB*, with different multi-core CPU architectures is an important contribution of this thesis.

This holistic exploration involves exploring the problem from different dimensions. It includes identification of the possible problematic areas in parallelized query execution from the perspective of operator's implementation and scheduling, plan parallelization techniques, effect of different workloads, dependence on the underlying hardware architecture, etc. The detailed experimentation and analysis provided in this thesis should act as a reference for any future explorations of analytical parallelized query execution in columnar database systems.

Appendix A

Sample TPC-H query graph visualizations

We list a few selected query execution data flow graphs in this section to give a perspective of the complexity of the execution plans when parallelized using static parallelization heuristic in MonetDB. A point to note is as the database system continuously evolves with better optimizer choices, efficient operator implementations, the plans tend to become more compact, resulting in less complex graphs. The rectangles represent operators while the edges represent the data-flow. The aim here is to show the complexity of the query plans in terms of their data-flow graph representation, without details about individual operators. Many of the operators are administrative operators, which have negligible cost, however, need to be present for column store specific data flow dependencies.

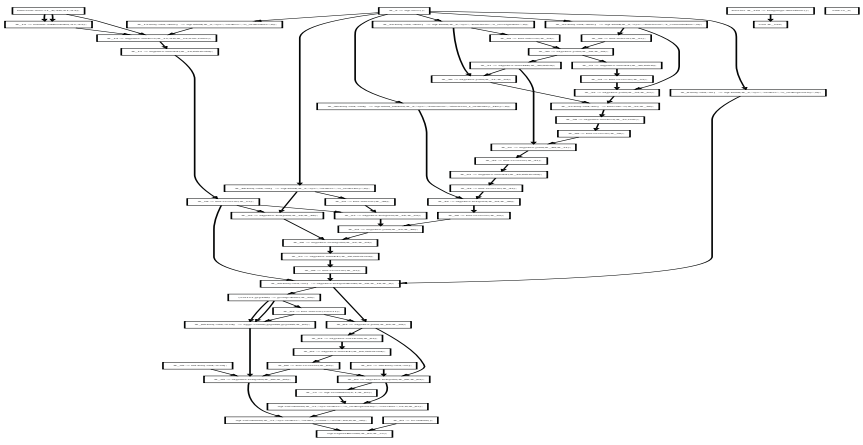

Figure A.1: **Query 4, 55 nodes.**

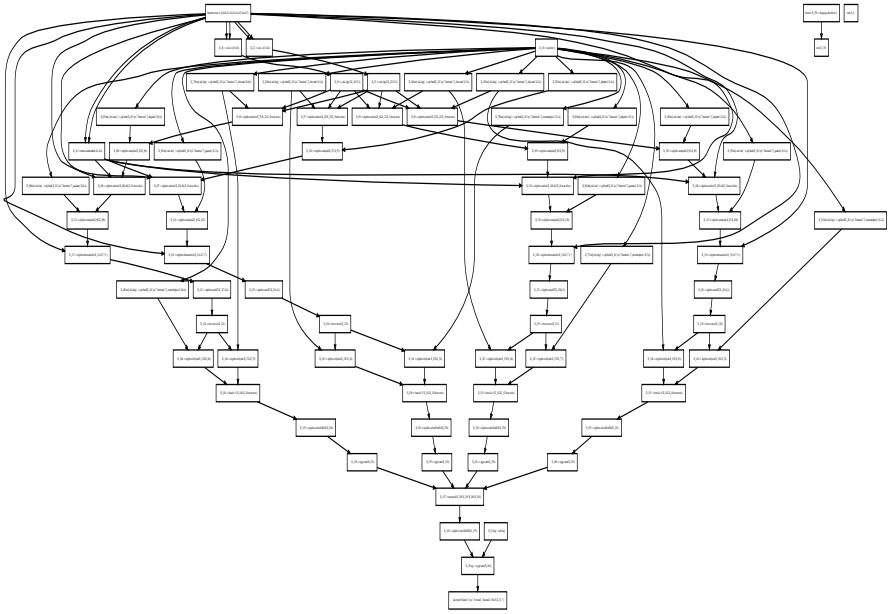

Figure A.2: **Query 6, 80 nodes.**

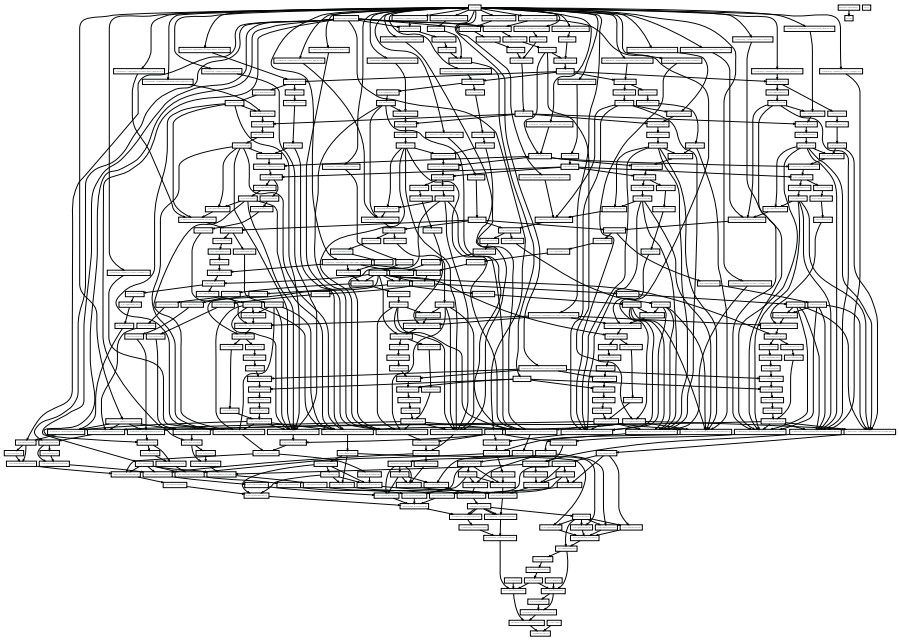

Figure A.3: **Query 8, 220 nodes.**

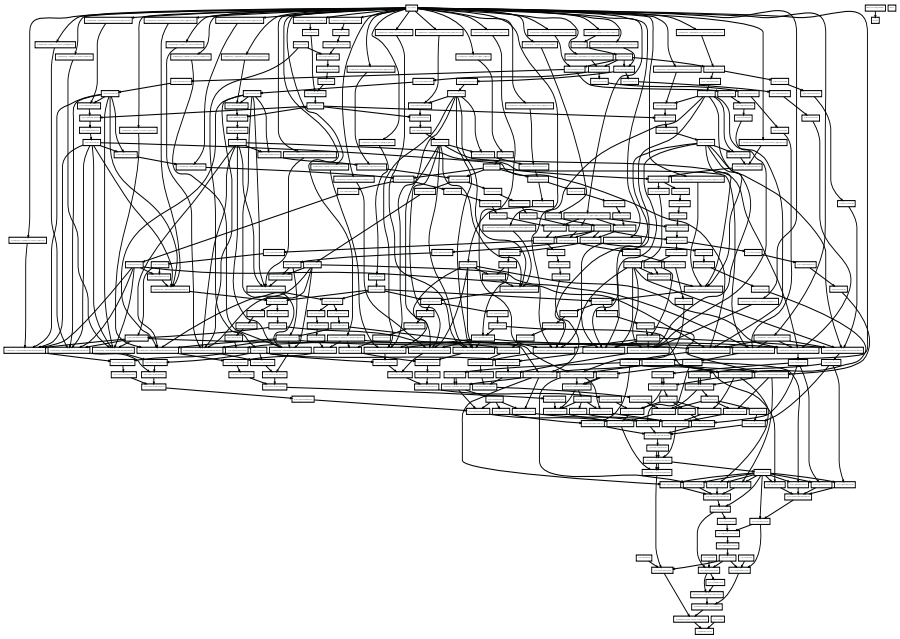

Figure A.4: **Query 9, 230 nodes.**

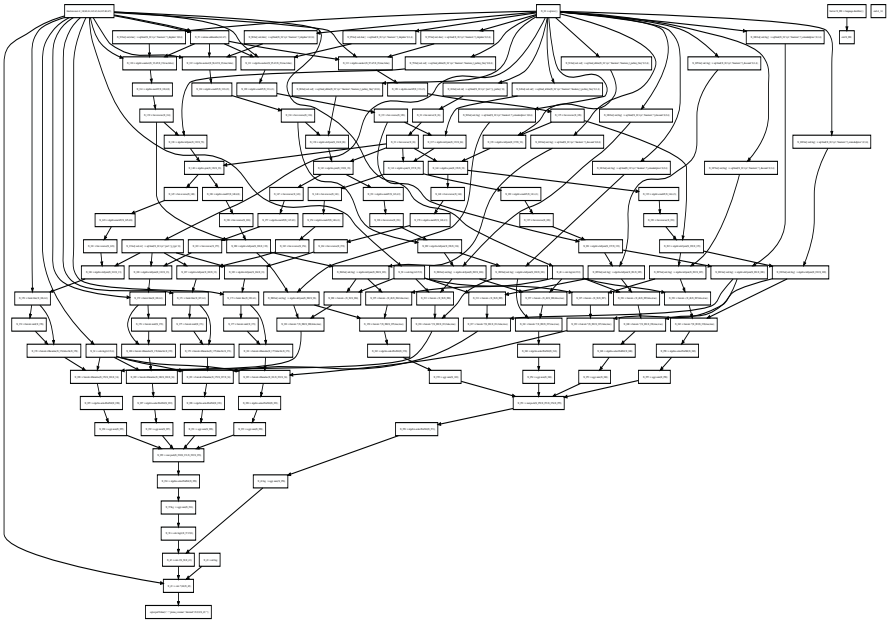

Figure A.5: **Query 14, 130 nodes.**

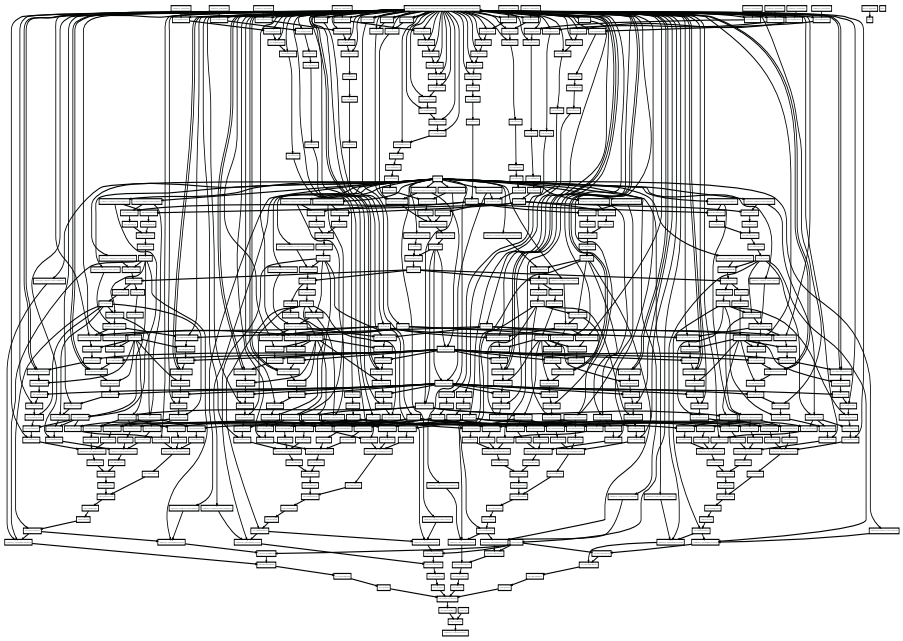

Figure A.6: **Query 19, 390 nodes.**

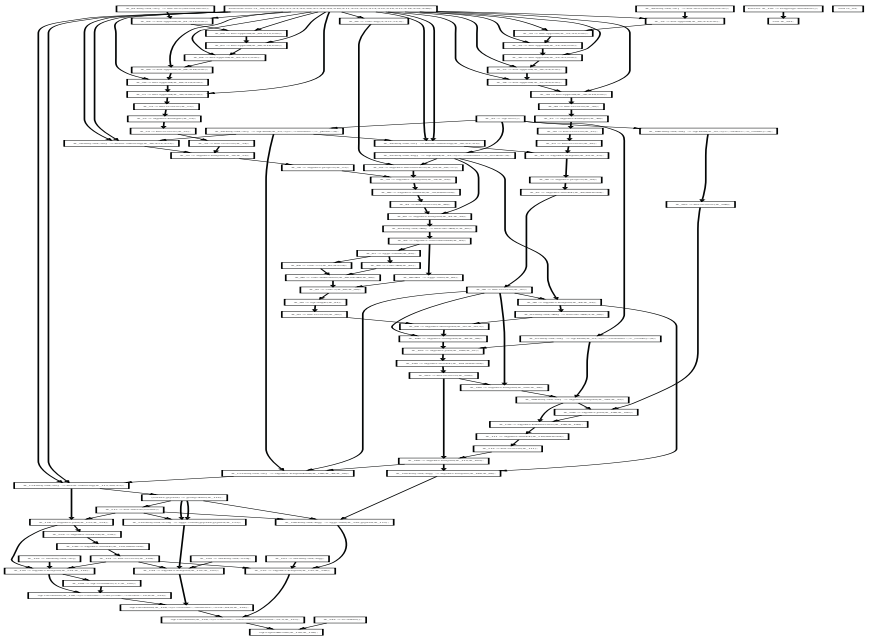

Figure A.7: **Query 22, 100 nodes.**

Appendix B

Sample TPC-DS Queries

This section lists sample TPC-DS queries used in Chapter 5 during adaptive parallelization experiments.

Q1

```
select count(*)
from store_sales, household_demographics, time_dim, store, web_sales
where
ss_sold_time_sk = time_dim.t_time_sk
and ss_hdemo_sk = household_demographics.hd_demo_sk
and ss_store_sk = s_store_sk
and time_dim.t_hour = 8
and time_dim.t_minute >= 30
and household_demographics.hd_dep_count = 5
and store.s_store_name = 'ese'
and ws_sold_date_sk >2451826
limit 10;
```

Q2

```
select count(*)
from store_returns, catalog_returns
where
sr_returned_date_sk = cr_returned_date_sk
and cr_returned_time_sk = sr_return_time_sk
and sr_fee = cr_fee
and sr_return_ship_cost = cr_return_ship_cost
and sr_return_ship_cost >500
and sr_item_sk=52457;
```

Q3

```
select count(*)
from customer_address, item, store, date_dim, promotion, customer, web_sales,
store_sales
```

```
where
ws_sold_date_sk = 2451826
and ss_sold_date_sk = 2451207
and ca_gmt_offset = -16
and i_category = 'Jewelry'
and s_gmt_offset = -6
and d_year = 2000
and p_cost = 600
and c_current_hdemo_sk >6000
limit 10;
```

Q4

```
select sum(cs_ext_discount_amt) as "excess discount amount"
from catalog_sales, item, date_dim
where i_manufact_id = 291
and i_item_sk = cs_item_sk

and d_date between date '2003-03-22'
and date '2003-11-14'
and d_date_sk = cs_sold_date_sk;
```

Q5

```
select sum(i_current_price)
from item, inventory, date_dim, catalog_sales
where i_current_price between 42 and 72
and inv_item_sk = i_item_sk
and d_date_sk=inv_date_sk
and d_date between date '2002-01-18' and date '2002-03-18'
and i_manufact_id in (744,691,853,946)
and inv_quantity_on_hand between 100 and 500
and cs_item_sk = i_item_sk;
```

Bibliography

[1] Actian vectorwise technical white paper. http://www.actian.com/media/whitepapers/unsorted/ivw-technical-wp.pdf.

[2] Adaptive query parallelization in multi-core column stores- under review sigmod 2016. https://sites.google.com/site/confproceed/sig-alternate.pdf.

[3] Adaptive query parallelization in multi-core column stores- under submission vldb 2015. https://sites.google.com/site/mrunalgawade/AdPar.pdf.

[4] co-operative scans. http://homepages.cwi.nl/ boncz/msc/2011-MichalSwitakowski.pdf.

[5] Dbtools. http://www.en.wikipedia.org/wiki/\Comparison_of_database_tools.

[6] Dot. www.graphviz.org/doc/info/lang.html.

[7] Graphviz. www.graphviz.org.

[8] Intel pcm tool. https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[9] Intel qpi. http://www.intel.nl/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[10] Monetdb. www.monetdb.org.

[11] Monetdb profiler. http://www.monetdb.org/Documentation/Manuals/MonetDB/Profiler.

[12] Multi-core query parallelism under resource contention - to be submitted to vldb 2016. https://sites.google.com/site/confproceed/AdParConcur.pdf.

[13] Mysql. "http://www.mysql.com".

[14] Numa scheduling progress in linux. http://lwn.net/Articles/568870/.

[15] Numactl. http://linux.die.net/man/8/numactl.

[16] Pgtune. http://pgtune.leopard.in.ua/.

[17] postgresvisualizer. http://www.dbplanview.com.

[18] Sql server management studio. http://www.msdn.microsoft.com/en-us/library/ms178071(v=sql.105).aspx.

[19] Sql server parallelization. "http://www.simple-talk.com/sql/learn-sql-server/understanding-and-using-parallelism-in-sql-server/".

[20] vectormulticore. http://homepages.cwi.nl/ boncz/msc/2010-KamilAnikijej.pdf.

[21] Zgrviewer. www.zvtm.sourceforge.net/zgrviewer.html.

[22] Zvtm. www.zvtm.sourceforge.net.

[23] Tableau data engine. "http://www.tableausoftware.com/sites/default/files/pages/tcc11-fastdataengine.pdf", 2010.

[24] M. Ahmad, A. Aboulnaga, and S. Babu. Query interactions in database workloads. In *Proceedings of the Second International Workshop on Testing Database Systems*, page 11. ACM, 2009.

[25] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Modeling and exploiting query interactions in database systems. In *Proc of CIKM*, pages 183–192. ACM, 2008.

[26] M. Ahmad et al. Qshuffler: Getting the query mix right. In *Proc of ICDE*, pages 1415–1417, 2008.

[27] A. Ailamaki, D. DeWitt, M. Hill, D. Wood, et al. Dbmss on a modern processor: Where does time go? In *Proc of VLDB*, pages 266–277, 1999.

[28] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[29] A. Aljanaby, E. Abuelrub, and M. Odeh. A survey of distributed query optimization. *Int. Arab J. Inf. Technol.*, 2(1):48–57, 2005.

[30] K. Anikiej. Multi-core parallelization of vectorized query execution. http://homepages.cwi.nl/ boncz/msc/2010-KamilAnikijej.pdf.

[31] R. Appuswamy, C. Gkantsidis, et al. Nobody ever got fired for buying a cluster. Technical report, Microsoft Technical Report MSR-TR-2013, 2013.

[32] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373. IEEE, 2013.

[33] A. Baumann et al. The multikernel: a new os architecture for scalable multicore systems. In *Proc of SIGOPS*, pages 29–44. ACM, 2009.

[34] S. Bellamkonda et al. Adaptive and big data scale parallel execution in oracle. *Proc of VLDB*, 6(11):1102–1113, 2013.

[35] S. Blair-Chappell and A. Stokes. *Parallel programming with intel parallel studio XE*. John Wiley & Sons, 2012.

[36] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 37–48. ACM, 2011.

[37] S. Blanas and J. M. Patel. How efficient is our radix join implementation, 2011.

[38] R. D. Blumofe et al. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[39] P. Boncz et al. Database architecture optimized for the new bottleneck: Memory access. In *Proc of VLDB*, pages 54–65, 1999.

[40] P. Boncz and M. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.

[41] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[42] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[43] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[44] S. Boyd-Wickizer, R. Morris, M. F. Kaashoek, et al. Reinventing scheduling for multicore systems. In *HotOS*, 2009.

[45] I. Chan. Oracle database performance tuning guide, 11g release 1 (11.1) b28274-02.

[46] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[47] X. Cheng, B. He, M. Lu, C. T. Lau, H. P. Huynh, and R. S. M. Goh. Efficient query processing on many-core architectures: A case study with intel xeon phi processor.

[48] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[49] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *Proc of VLDB*, pages 339–350. VLDB Endowment, 2007.

[50] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *Proc of the DAMON*, pages 25–34. ACM, 2008.

[51] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[52] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[53] S. Dam and G. Fritchey. *SQL Server 2008 Query Performance Tuning Distilled*. Apress, 2009.

[54] A. Damico. Transitioning to r: Replicating sas, stata, and sudaan analysis techniques in health policy data.

[55] U. Dayal et al. Managing operational business intelligence workloads. *Proc of SIGOPS*, pages 92–98, 2009.

[56] A. C. de Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, 2010.

[57] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[58] A. Deshpande et al. Adaptive query processing: why, how, when, what next? In *Proc of VLDB*, pages 1426–1427, 2007.

[59] D. J. DeWitt and R. Gerber. *Multiprocessor hash-based join algorithms*. University of Wisconsin-Madison, Computer Sciences Department, 1985.

[60] A. Ganapathi et al. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc of ICDE*, pages 592–603. IEEE, 2009.

[61] S. Ganguly et al. Query optimization for parallel execution. In *ACM SIGMOD Record*, volume 21, pages 9–18, 1992.

[62] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot. Technical report, Technical report, AT&T Research. URL http://www. graphviz. org/Documentation/dotguide. pdf, 2006.

[63] M. Gawade and M. Kersten. Stethoscope: a platform for interactive visual analysis of query execution plans. *Proc of VLDB*, 5:1926–1929, 2012.

[64] M. Gawade and M. Kersten. Tomograph: Highlighting query parallelism in a multi-core system. In *Proc of DBTest*, page 3. ACM, 2013.

[65] M. Gawade and M. Kersten. Numa obliviousness through memory mapping. In *Proc of DAMON*, page 7, 2015.

[66] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: killing one thousand queries with one stone. *Proc of VLDB*, 5(6):526–537, 2012.

[67] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *Proceedings of the VLDB Endowment*, 8(3):233–244, 2014.

[68] G. Graefe. *Encapsulation of parallelism in the Volcano query processing system*, volume 19. ACM, 1990.

[69] G. Graefe. Sort-merge-join: an idea whose time has (h) passed? In *Data Engineering, 1994. Proceedings. 10th International Conference*, pages 406–417. IEEE, 1994.

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

[71] G. Graefe et al. Robust Query Processing (Dagstuhl Seminar 12321). *Dagstuhl Reports*, 2(8):1–15, 2012.

[72] G. Graefe, H. Kuno, and J. Wiener. Visualizing the robustness of query execution. *arXiv preprint arXiv:0909.1772*, 2009.

[73] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010.

[74] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[75] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. *Proceedings of the VLDB Endowment*, 1(1):188–200, 2008.

[76] N. Hardavellas et al. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, volume 7, pages 79–87. Citeseer, 2007.

[77] S. Harizopoulos and A. Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[78] S. Harizopoulos et al. Qpipe: a simultaneously pipelined relational query engine. In *Proc of SIGMOD*, pages 383–394. ACM, 2005.

[79] T. Harris, M. Maas, and V. J. Marathe. Callisto: co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, page 24. ACM, 2014.

[80] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[81] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd international workshop on Data management on new hardware*, page 4. ACM, 2007.

[82] W. Hong. *Parallel query processing using shared memory multiprocessors*. PhD thesis, UC, Berkeley, 1992.

[83] B. Howe, G. Cole, E. Souroush, P. Koutris, A. Key, N. Khoussainova, and L. Battle. Database-as-a-service for long-tail science. In *International Conference on Scientific and Statistical Database Management*, pages 480–489. Springer, 2011.

[84] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proc of SIGMOD*, pages 297–308. ACM, 2009.

[85] S. Idreos, M. L. Kersten, S. Manegold, et al. Database cracking. In *CIDR*, volume 3, pages 1–8, 2007.

[86] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[87] P. N. R. Jadhav. Performance evaluation of oracle parallel execution. Master's thesis, California State University, Sacramento, 2011.

[88] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

[89] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *Proceedings of the VLDB Endowment*, 8(6):642–653, 2015.

[90] R. Johnson et al. To share or not to share? In *Proc of VLDB*, pages 351–362. VLDB Endowment, 2007.

[91] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35. ACM, 2009.

[92] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD Record*, volume 27, pages 106–117. ACM, 1998.

[93] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[94] M. L. Kersten, S. Plomp, and C. A. van den Berg. Object storage management in goblin. 1992.

[95] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[96] S. Krompass et al. Dynamic workload management for very large data warehouses: Juggling feathers and bowling balls. In *Proc of VLDB*, pages 1105–1115. VLDB Endowment, 2007.

[97] S. Krompass et al. Managing long-running queries. In *Proc of EDBT*, pages 132–143. ACM, 2009.

[98] S. Krompass, A. Scholz, M.-C. Albutiu, H. A. Kuno, J. L. Wiener, U. Dayal, and A. Kemper. Quality of service-enabled management of database workloads. *IEEE Data Eng. Bull.*, 31(1):20–27, 2008.

[99] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proc of VLDB*, 5(12):1790–1801, 2012.

[100] C. Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40, 2013.

[101] R. S. Lanzelotte et al. On the effectiveness of optimization search strategies for parallel execution spaces. In *VLDB*, volume 93, pages 493–504, 1993.

[102] P.-Å. Larson et al. Sql server column store indexes. In *Proc of Sigmod*, pages 1177–1184, 2011.

[103] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

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

[105] B. Lewis, D. Berg, et al. *Multithreaded programming with Pthreads*, volume 2550. Sun Microsystems Press, 1998.

[106] Y. Li et al. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[107] Z. Majo et al. Memory system performance in a numa multicore multiprocessor. In *Proc of ICSS*, page 12, 2011.

[108] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *Knowledge and Data Engineering, IEEE Transactions on*, 14(4):709–730, 2002.

[109] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-conscious radix-decluster projections. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 684–695. VLDB Endowment, 2004.

[110] S. Manegold et al. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *Proc of VLDB*, 2(2):1648–1653, 2009.

[111] C. McCurdy et al. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Proc of ISPASS*, pages 87–96, 2010.

[112] A. Mehta et al. Bi batch manager: a system for managing batch workloads on enterprise data-warehouses. In *Proc of EDBT, 640-651*, 2008.

[113] C. Mohan. Impact of recent hardware and software trends on high performance transaction processing and analytics. In *Proc of PEMCCS*, pages 85–92. Springer, 2011.

[114] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proc of HPCMP*, pages 7–10, 1999.

[115] V. R. Narasayya et al. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *Proc of CIDR*, 2013.

[116] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[117] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[118] E. Pietriga. A toolkit for addressing hci issues in visual language environments. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 145–152. IEEE, 2005.

[119] H. Pirk et al. Database cracking: Fancy scan, not poor man's sort! In *Proc of the DaMon*, 2014.

[120] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 508–519. IEEE, 2014.

[121] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.

[122] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. Oltp on hardware islands. *Proceedings of the VLDB Endowment*, 5(11):1447–1458, 2012.

[123] R. Ramakrishnan and J. Gehrke. Database management systems. 2000.

[124] V. Raman et al. Db2 with blu acceleration: So much more than just a column store. *Proc of VLDB*, pages 1080–1091, 2013.

[125] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. IEEE, 2007.

[126] J. Reinders. *VTune performance analyzer essentials*. Intel Press, 2005.

[127] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.

[128] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proc of SIGMOD*, pages 1231–1242, 2013.

[129] R. A. Rutenbar. Simulated annealing algorithms: An overview. *Circuits and Devices, IEEE*, 5(1):19–26, 1989.

[130] M. Saecker and V. Markl. Big data analytics on modern hardware architectures: A technology survey. In *Business Intelligence*, pages 125–149. Springer, 2013.

[131] T.-I. Salomie et al. Database engines on multicores, why parallelize when you can distribute? In *Proc of Eurosys*, pages 17–30, 2011.

[132] D. Scheibli, C. Dinse, and A. Boehm. Qe3d: Interactive visualization and exploration of complex, distributed query plans. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 877–881. ACM, 2015.

[133] B. Schroeder et al. How to determine a good multi-programming level for external scheduling. In *Proc of ICDE*, pages 60–60. IEEE, 2006.

[134] P. G. Selinger, M. M. Astrahan, Chamberlin, et al. Access path selection in a relational database management system. In *Proc of SIGMOD*, pages 23–34, 1979.

[135] D. E. Shasha and P. Bonnet. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann, 2003.

[136] L. Sidirourgos and M. Kersten. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 893–904. ACM, 2013.

[137] A. Simitsis, K. Wilkinson, J. Blais, and J. Walsh. Vqa: vertica query analyzer. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 701–704. ACM, 2014.

[138] M. Stillger, G. M. Lohman, V. Markl, et al. Leo-db2's learning optimizer. In *Proc of VLDB*, pages 19–28, 2001.

[139] M. Stonebraker et al. *The design of Postgres*, volume 15. ACM, 1986.

[140] S. Tozer et al. Q-cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *Proc of ICDE*, pages 397–408. IEEE, 2010.

[141] P. Valduriez. Parallel database systems: Open problems and new issues. *Distributed and parallel Databases*, 1(2):137–165, 1993.

[142] L. Viktor et al. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proc of SIGMOD*, 2014.

[143] R. Wesley et al. An analytic data engine for visualization in tableau. In *Proc of SIGMOD*, pages 1185–1194. ACM, 2011.

[144] J. L. Wiener, H. Kuno, and G. Graefe. Benchmarking query execution robustness. In *Performance Evaluation and Benchmarking*, pages 153–166. Springer, 2009.

[145] S. Zhang, J. He, B. He, and M. Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proceedings of the VLDB Endowment*, 6(12):1374–1377, 2013.

[146] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *Proc of VLDB*, pages 49–60. VLDB Endowment, 2005.

[147] M. Zukowski, S. Héman, et al. Cooperative scans: dynamic bandwidth sharing in a dbms. In *Proc of VLDB*, pages 723–734, 2007.

List of Figures

List of Tables

SIKS Dissertations

2009

2009-01 Rasa Jurgelenaite (RUN), *S*ymmetric Causal Independence Models.
2009-02 Willem Robert van Hage (VU), *E*valuating Ontology-Alignment Techniques.
2009-03 Hans Stol (UvT), *A* Framework for Evidence-based Policy Making Using IT.
2009-04 Josephine Nabukenya (RUN), *I*mproving the Quality of Organisational Policy Making using Collaboration Engineering.
2009-05 Sietse Overbeek (RUN), *B*ridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality.
2009-06 Muhammad Subianto (UU), *U*nderstanding Classification.
2009-07 Ronald Poppe (UT), *D*iscriminative Vision-Based Recovery and Recognition of Human Motion.
2009-08 Volker Nannen (VU), *E*volutionary Agent-Based Policy Analysis in Dynamic Environments.
2009-09 Benjamin Kanagwa (RUN), *D*esign, Discovery and Construction of Service-oriented Systems.
2009-10 Jan Wielemaker (UVA), *L*ogic programming for knowledge-intensive interactive applications.
2009-11 Alexander Boer (UVA), *L*egal Theory, Sources of Law & the Semantic Web.
2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin), *O*perating Guidelines for Services.
2009-13 Steven de Jong (UM), *F*airness in Multi-Agent Systems.
2009-14 Maksym Korotkiy (VU), *F*rom ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA).
2009-15 Rinke Hoekstra (UVA), *O*ntology Representation - Design Patterns and Ontologies that Make Sense.
2009-16 Fritz Reul (UvT), *N*ew Architectures in Computer Chess.
2009-17 Laurens van der Maaten (UvT), *F*eature Extraction from Visual Data.
2009-18 Fabian Groffen (CWI), *A*rmada, An Evolving Database System.
2009-19 Valentin Robu (CWI), *M*odeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets.
2009-20 Bob van der Vecht (UU), *A*djustable Autonomy: Controling Influences on Decision Making.
2009-21 Stijn Vanderlooy (UM), *R*anking and Reliable Classification.
2009-22 Pavel Serdyukov (UT), *S*earch For Expertise: Going beyond direct evidence.
2009-23 Peter Hofgesang (VU), *M*odelling Web Usage in a Changing Environment.
2009-24 Annerieke Heuvelink (VUA), *C*ognitive Models for Training Simulations.
2009-25 Alex van Ballegooij (CWI), "RAM: Array Database Management through Relational Mapping".
2009-26 Fernando Koch (UU), *A*n Agent-Based Model for the Development of Intelligent Mobile Services.
2009-27 Christian Glahn (OU), *C*ontextual Support of social Engagement and Reflection on the Web.
2009-28 Sander Evers (UT), *S*ensor Data Management with Probabilistic Models.
2009-29 Stanislav Pokraev (UT), *M*odel-Driven Semantic Integration of Service-Oriented Applications.
2009-30 Marcin Zukowski (CWI), *B*alancing vectorized query execution with bandwidth-optimized storage.
2009-31 Sofiya Katrenko (UVA), *A* Closer Look at Learning Relations from Text.
2009-32 Rik Farenhorst (VU) and Remco de Boer (VU), *A*rchitectural Knowledge Management: Supporting Architects and Auditors.
2009-33 Khiet Truong (UT), *H*ow Does Real Affect Affect Affect Recognition In Speech?.
2009-34 Inge van de Weerd (UU), *A*dvancing in Software Product Management: An Incremental Method Engineering Approach.
2009-35 Wouter Koelewijn (UL), *P*rivacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling.
2009-36 Marco Kalz (OUN), *P*lacement Support for Learners in Learning Networks.
2009-37 Hendrik Drachsler (OUN), *N*avigation Support for Learners in Informal Learning Networks.
2009-38 Riina Vuorikari (OU), *T*ags and self-organisation: a metadata ecology for learning resources in a multilingual context.
2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin), *S*ervice Substitution – A Behavioral

Approach Based on Petri Nets.
2009-40 Stephan Raaijmakers (UvT), *M*ultinomial Language Learning: Investigations into the Geometry of Language.
2009-41 Igor Berezhnyy (UvT), *D*igital Analysis of Paintings.
2009-42 Toine Bogers (UvT), *R*ecommender Systems for Social Bookmarking.
2009-43 Virginia Nunes Leal Franqueira (UT), *F*inding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients.
2009-44 Roberto Santana Tapia (UT), *A*ssessing Business-IT Alignment in Networked Organizations.
2009-45 Jilles Vreeken (UU), *M*aking Pattern Mining Useful.
2009-46 Loredana Afanasiev (UvA), *Q*uerying XML: Benchmarks and Recursion.

2010

2010-01 Matthijs van Leeuwen (UU), *P*atterns that Matter.
2010-02 Ingo Wassink (UT), *W*ork flows in Life Science.
2010-03 Joost Geurts (CWI), *A* Document Engineering Model and Processing Framework for Multimedia documents.
2010-04 Olga Kulyk (UT), *D*o You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments.
2010-05 Claudia Hauff (UT), *P*redicting the Effectiveness of Queries and Retrieval Systems.
2010-06 Sander Bakkes (UvT), *R*apid Adaptation of Video Game AI.
2010-07 Wim Fikkert (UT), *G*esture interaction at a Distance.
2010-08 Krzysztof Siewicz (UL), *T*owards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments.
2010-09 Hugo Kielman (UL), *A* Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging.
2010-10 Rebecca Ong (UL), *M*obile Communication and Protection of Children.
2010-11 Adriaan Ter Mors (TUD), *T*he world according to MARP: Multi-Agent Route Planning.
2010-12 Susan van den Braak (UU), *S*ensemaking software for crime analysis.
2010-13 Gianluigi Folino (RUN), *H*igh Performance Data Mining using Bio-inspired techniques.
2010-14 Sander van Splunter (VU), *A*utomated Web Service Reconfiguration.
2010-15 Lianne Bodenstaff (UT), *M*anaging Dependency Relations in Inter-Organizational Models.
2010-16 Sicco Verwer (TUD), *E*fficient Identification of Timed Automata, theory and practice.
2010-17 Spyros Kotoulas (VU), *S*calable Discovery of Networked Resources: Algorithms, Infrastructure, Applications.
2010-18 Charlotte Gerritsen (VU), *C*aught in the Act: Investigating Crime by Agent-Based Simulation.
2010-19 Henriette Cramer (UvA), *P*eople's Responses to Autonomous and Adaptive Systems.
2010-20 Ivo Swartjes (UT), *W*hose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative.
2010-21 Harold van Heerde (UT), *P*rivacy-aware data management by means of data degradation.
2010-22 Michiel Hildebrand (CWI), *E*nd-user Support for Access to Heterogeneous Linked Data.
2010-23 Bas Steunebrink (UU), *T*he Logical Structure of Emotions.
2010-24 Dmytro Tykhonov, *D*esigning Generic and Efficient Negotiation Strategies.
2010-25 Zulfiqar Ali Memon (VU), *M*odelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective.
2010-26 Ying Zhang (CWI), *X*RPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines.
2010-27 Marten Voulon (UL), *A*utomatisch contracteren.
2010-28 Arne Koopman (UU), *C*haracteristic Relational Patterns.
2010-29 Stratos Idreos(CWI), *D*atabase Cracking: Towards Auto-tuning Database Kernels.
2010-30 Marieke van Erp (UvT), *A*ccessing Natural History - Discoveries in data cleaning, structuring, and retrieval.
2010-31 Victor de Boer (UVA), *O*ntology Enrichment from Heterogeneous Sources on the Web.
2010-32 Marcel Hiel (UvT), *A*n Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems.

2010-33 Robin Aly (UT), *M*odeling Representation Uncertainty in Concept-Based Multimedia Retrieval.
2010-34 Teduh Dirgahayu (UT), *I*nteraction Design in Service Compositions.
2010-35 Dolf Trieschnigg (UT), *P*roof of Concept: Concept-based Biomedical Information Retrieval.
2010-36 Jose Janssen (OU), *P*aving the Way for Lifelong Learning; Facilitating competence development through a learning path specification.
2010-37 Niels Lohmann (TUE), *C*orrectness of services and their composition.
2010-38 Dirk Fahland (TUE), *F*rom Scenarios to components.
2010-39 Ghazanfar Farooq Siddiqui (VU), *I*ntegrative modeling of emotions in virtual agents.
2010-40 Mark van Assem (VU), *C*onverting and Integrating Vocabularies for the Semantic Web.
2010-41 Guillaume Chaslot (UM), *M*onte-Carlo Tree Search.
2010-42 Sybren de Kinderen (VU), *N*eeds-driven service bundling in a multi-supplier setting - the computational e3-service approach.
2010-43 Peter van Kranenburg (UU), *A* Computational Approach to Content-Based Retrieval of Folk Song Melodies.
2010-44 Pieter Bellekens (TUE), *A*n Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain.
2010-45 Vasilios Andrikopoulos (UvT), *A* theory and model for the evolution of software services.
2010-46 Vincent Pijpers (VU), *e*3alignment: Exploring Inter-Organizational Business-ICT Alignment.
2010-47 Chen Li (UT), *M*ining Process Model Variants: Challenges, Techniques, Examples.
2010-48 Withdrawn, .
2010-49 Jahn-Takeshi Saito (UM), *S*olving difficult game positions.
2010-50 Bouke Huurnink (UVA), *S*earch in Audiovisual Broadcast Archives.
2010-51 Alia Khairia Amin (CWI), *U*nderstanding and supporting information seeking tasks in multiple sources.
2010-52 Peter-Paul van Maanen (VU), *A*daptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention.
2010-53 Edgar Meij (UVA), *C*ombining Concepts and Language Models for Information Access.

2011

2011-01 Botond Cseke (RUN), *V*ariational Algorithms for Bayesian Inference in Latent Gaussian Models.
2011-02 Nick Tinnemeier(UU), *O*rganizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language.
2011-03 Jan Martijn van der Werf (TUE), *C*ompositional Design and Verification of Component-Based Information Systems.
2011-04 Hado van Hasselt (UU), *I*nsights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference.
2011-05 Base van der Raadt (VU), *E*nterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline..
2011-06 Yiwen Wang (TUE), *S*emantically-Enhanced Recommendations in Cultural Heritage.
2011-07 Yujia Cao (UT), *M*ultimodal Information Presentation for High Load Human Computer Interaction.
2011-08 Nieske Vergunst (UU), *B*DI-based Generation of Robust Task-Oriented Dialogues.
2011-09 Tim de Jong (OU), *C*ontextualised Mobile Media for Learning.
2011-10 Bart Bogaert (UvT), *C*loud Content Contention.
2011-11 Dhaval Vyas (UT), *D*esigning for Awareness: An Experience-focused HCI Perspective.
2011-12 Carmen Bratosin (TUE), *G*rid Architecture for Distributed Process Mining.
2011-13 Xiaoyu Mao (UvT), *A*irport under Control. Multiagent Scheduling for Airport Ground Handling.
2011-14 Milan Lovric (EUR), *B*ehavioral Finance and Agent-Based Artificial Markets.
2011-15 Marijn Koolen (UvA), *T*he Meaning of Structure: the Value of Link Evidence for Information Retrieval.
2011-16 Maarten Schadd (UM), *S*elective Search in Games of Different Complexity.
2011-17 Jiyin He (UVA), *E*xploring Topic Structure: Coherence, Diversity and Relatedness.

2011-18 Mark Ponsen (UM), *S*trategic Decision-Making in complex games.
2011-19 Ellen Rusman (OU), *T*he Mind ' s Eye on Personal Profiles.
2011-20 Qing Gu (VU), *G*uiding service-oriented software engineering - A view-based approach.
2011-21 Linda Terlouw (TUD), *M*odularization and Specification of Service-Oriented Systems.
2011-22 Junte Zhang (UVA), *S*ystem Evaluation of Archival Description and Access.
2011-23 Wouter Weerkamp (UVA), *F*inding People and their Utterances in Social Media.
2011-24 Herwin van Welbergen (UT), *B*ehavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior.
2011-25 Syed Waqar ul Qounain Jaffry (VU)), *A*nalysis and Validation of Models for Trust Dynamics.
2011-26 Matthijs Aart Pontier (VU), *V*irtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots.
2011-27 Aniel Bhulai (VU), *D*ynamic website optimization through autonomous management of design patterns.
2011-28 Rianne Kaptein(UVA), *E*ffective Focused Retrieval by Exploiting Query Context and Document Structure.
2011-29 Faisal Kamiran (TUE), *D*iscrimination-aware Classification.
2011-30 Egon van den Broek (UT), *A*ffective Signal Processing (ASP): Unraveling the mystery of emotions.
2011-31 Ludo Waltman (EUR), *C*omputational and Game-Theoretic Approaches for Modeling Bounded Rationality.
2011-32 Nees-Jan van Eck (EUR), *M*ethodological Advances in Bibliometric Mapping of Science.
2011-33 Tom van der Weide (UU), *A*rguing to Motivate Decisions.
2011-34 Paolo Turrini (UU), *S*trategic Reasoning in Interdependence: Logical and Game-theoretical Investigations.
2011-35 Maaike Harbers (UU), *E*xplaining Agent Behavior in Virtual Training.
2011-36 Erik van der Spek (UU), *E*xperiments in serious game design: a cognitive approach.
2011-37 Adriana Burlutiu (RUN), *M*achine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference.
2011-38 Nyree Lemmens (UM), *B*ee-inspired Distributed Optimization.
2011-39 Joost Westra (UU), *O*rganizing Adaptation using Agents in Serious Games.
2011-40 Viktor Clerc (VU), *A*rchitectural Knowledge Management in Global Software Development.
2011-41 Luan Ibraimi (UT), *C*ryptographically Enforced Distributed Data Access Control.
2011-42 Michal Sindlar (UU), *E*xplaining Behavior through Mental State Attribution.
2011-43 Henk van der Schuur (UU), *P*rocess Improvement through Software Operation Knowledge.
2011-44 Boris Reuderink (UT), *R*obust Brain-Computer Interfaces.
2011-45 Herman Stehouwer (UvT), *S*tatistical Language Models for Alternative Sequence Selection.
2011-46 Beibei Hu (TUD), *T*owards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work.
2011-47 Azizi Bin Ab Aziz(VU), *E*xploring Computational Models for Intelligent Support of Persons with Depression.
2011-48 Mark Ter Maat (UT), *R*esponse Selection and Turn-taking for a Sensitive Artificial Listening Agent.
2011-49 Andreea Niculescu (UT), *C*onversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality.

2012

2012-01 Terry Kakeeto (UvT), *R*elationship Marketing for SMEs in Uganda.
2012-02 Muhammad Umair(VU), *A*daptivity, emotion, and Rationality in Human and Ambient Agent Models.
2012-03 Adam Vanya (VU), *S*upporting Architecture Evolution by Mining Software Repositories.
2012-04 Jurriaan Souer (UU), *D*evelopment of Content Management System-based Web Applications.
2012-05 Marijn Plomp (UU), *M*aturing Interorganisational Information Systems.
2012-06 Wolfgang Reinhardt (OU), *A*wareness Support for Knowledge Workers in Research Networks.
2012-07 Rianne van Lambalgen (VU), *W*hen the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions.

2012-08 Gerben de Vries (UVA), *K*ernel Methods for Vessel Trajectories.
2012-09 Ricardo Neisse (UT), *T*rust and Privacy Management Support for Context-Aware Service Platforms.
2012-10 David Smits (TUE), *T*owards a Generic Distributed Adaptive Hypermedia Environment.
2012-11 J.C.B. Rantham Prabhakara (TUE), *P*rocess Mining in the Large: Preprocessing, Discovery, and Diagnostics.
2012-12 Kees van der Sluijs (TUE), *M*odel Driven Design and Data Integration in Semantic Web Information Systems.
2012-13 Suleman Shahid (UvT), *F*un and Face: Exploring non-verbal expressions of emotion during playful interactions.
2012-14 Evgeny Knutov(TUE), *G*eneric Adaptation Framework for Unifying Adaptive Web-based Systems.
2012-15 Natalie van der Wal (VU), *S*ocial Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes..
2012-16 Fiemke Both (VU), *H*elping people by understanding them - Ambient Agents supporting task execution and depression treatment.
2012-17 Amal Elgammal (UvT), *T*owards a Comprehensive Framework for Business Process Compliance.
2012-18 Eltjo Poort (VU), *I*mproving Solution Architecting Practices.
2012-19 Helen Schonenberg (TUE), *W*hat's Next? Operational Support for Business Process Execution.
2012-20 Ali Bahramisharif (RUN), *C*overt Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing.
2012-21 Roberto Cornacchia (TUD), *Q*uerying Sparse Matrices for Information Retrieval.
2012-22 Thijs Vis (UvT), *I*ntelligence, politie en veiligheidsdienst: verenigbare grootheden?.
2012-23 Christian Muehl (UT), *T*oward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction.
2012-24 Laurens van der Werff (UT), *E*valuation of Noisy Transcripts for Spoken Document Retrieval.
2012-25 Silja Eckartz (UT), *M*anaging the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application.
2012-26 Emile de Maat (UVA), *M*aking Sense of Legal Text.
2012-27 Hayrettin Gurkok (UT), *M*ind the Sheep! User Experience Evaluation & Brain-Computer Interface Games.
2012-28 Nancy Pascall (UvT), *E*ngendering Technology Empowering Women.
2012-29 Almer Tigelaar (UT), *P*eer-to-Peer Information Retrieval.
2012-30 Alina Pommeranz (TUD), *D*esigning Human-Centered Systems for Reflective Decision Making.
2012-31 Emily Bagarukayo (RUN), *A* Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure.
2012-32 Wietske Visser (TUD), *Q*ualitative multi-criteria preference representation and reasoning.
2012-33 Rory Sie (OUN), *C*oalitions in Cooperation Networks (COCOON).
2012-34 Pavol Jancura (RUN), *E*volutionary analysis in PPI networks and applications.
2012-35 Evert Haasdijk (VU), *N*ever Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics.
2012-36 Denis Ssebugwawo (RUN), *A*nalysis and Evaluation of Collaborative Modeling Processes.
2012-37 Agnes Nakakawa (RUN), *A* Collaboration Process for Enterprise Architecture Creation.
2012-38 Selmar Smit (VU), *P*arameter Tuning and Scientific Testing in Evolutionary Algorithms.
2012-39 Hassan Fatemi (UT), *R*isk-aware design of value and coordination networks.
2012-40 Agus Gunawan (UvT), *I*nformation Access for SMEs in Indonesia.
2012-41 Sebastian Kelle (OU), *G*ame Design Patterns for Learning.
2012-42 Dominique Verpoorten (OU), *R*eflection Amplifiers in self-regulated Learning.
2012-43 Withdrawn, .
2012-44 Anna Tordai (VU), *O*n Combining Alignment Techniques.
2012-45 Benedikt Kratz (UvT), *A* Model and Language for Business-aware Transactions.
2012-46 Simon Carter (UVA), *E*xploration and Exploitation of Multilingual Data for Statistical Machine Translation.
2012-47 Manos Tsagkias (UVA), *M*ining Social Media: Tracking Content and Predicting Behavior.

2012-48 Jorn Bakker (TUE), *H*andling Abrupt Changes in Evolving Time-series Data.
2012-49 Michael Kaisers (UM), *L*earning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions.
2012-50 Steven van Kervel (TUD), *O*ntologogy driven Enterprise Information Systems Engineering.
2012-51 Jeroen de Jong (TUD), *H*euristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching.

2013

2013-01 Viorel Milea (EUR), *N*ews Analytics for Financial Decision Support.
2013-02 Erietta Liarou (CWI), *M*onetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing.
2013-03 Szymon Klarman (VU), *R*easoning with Contexts in Description Logics.
2013-04 Chetan Yadati(TUD), *C*oordinating autonomous planning and scheduling.
2013-05 Dulce Pumareja (UT), *G*roupware Requirements Evolutions Patterns.
2013-06 Romulo Goncalves(CWI), *T*he Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience.
2013-07 Giel van Lankveld (UvT), *Q*uantifying Individual Player Differences.
2013-08 Robbert-Jan Merk(VU), *M*aking enemies: cognitive modeling for opponent agents in fighter pilot simulators.
2013-09 Fabio Gori (RUN), *M*etagenomic Data Analysis: Computational Methods and Applications.
2013-10 Jeewanie Jayasinghe Arachchige(UvT), *A* Unified Modeling Framework for Service Design..
2013-11 Evangelos Pournaras(TUD), *M*ulti-level Reconfigurable Self-organization in Overlay Services.
2013-12 Marian Razavian(VU), *K*nowledge-driven Migration to Services.
2013-13 Mohammad Safiri(UT), *S*ervice Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly.
2013-14 Jafar Tanha (UVA), *E*nsemble Approaches to Semi-Supervised Learning Learning.
2013-15 Daniel Hennes (UM), *M*ultiagent Learning - Dynamic Games and Applications.
2013-16 Eric Kok (UU), *E*xploring the practical benefits of argumentation in multi-agent deliberation.
2013-17 Koen Kok (VU), *T*he PowerMatcher: Smart Coordination for the Smart Electricity Grid.
2013-18 Jeroen Janssens (UvT), *O*utlier Selection and One-Class Classification.
2013-19 Renze Steenhuizen (TUD), *C*oordinated Multi-Agent Planning and Scheduling.
2013-20 Katja Hofmann (UvA), *F*ast and Reliable Online Learning to Rank for Information Retrieval.
2013-21 Sander Wubben (UvT), *T*ext-to-text generation by monolingual machine translation.
2013-22 Tom Claassen (RUN), *C*ausal Discovery and Logic.
2013-23 Patricio de Alencar Silva(UvT), *V*alue Activity Monitoring.
2013-24 Haitham Bou Ammar (UM), *A*utomated Transfer in Reinforcement Learning.
2013-25 Agnieszka Anna Latoszek-Berendsen (UM), *I*ntention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System.
2013-26 Alireza Zarghami (UT), *A*rchitectural Support for Dynamic Homecare Service Provisioning.
2013-27 Mohammad Huq (UT), *I*nference-based Framework Managing Data Provenance.
2013-28 Frans van der Sluis (UT), *W*hen Complexity becomes Interesting: An Inquiry into the Information eXperience.
2013-29 Iwan de Kok (UT), *L*istening Heads.
2013-30 Joyce Nakatumba (TUE), *R*esource-Aware Business Process Management: Analysis and Support.
2013-31 Dinh Khoa Nguyen (UvT), *B*lueprint Model and Language for Engineering Cloud Applications.
2013-32 Kamakshi Rajagopal (OUN), *N*etworking For Learning; The role of Networking in a Lifelong Learner's Professional Development.
2013-33 Qi Gao (TUD), *U*ser Modeling and Personalization in the Microblogging Sphere.
2013-34 Kien Tjin-Kam-Jet (UT), *D*istributed Deep Web Search.
2013-35 Abdallah El Ali (UvA), *M*inimal Mobile Human Computer Interaction.
2013-36 Than Lam Hoang (TUe), *P*attern Mining in Data Streams.
2013-37 Dirk Börner (OUN), *A*mbient Learning Displays.
2013-38 Eelco den Heijer (VU), *A*utonomous Evolutionary Art.

2013-39 Joop de Jong (TUD), *A* Method for Enterprise Ontology based Design of Enterprise Information Systems.
2013-40 Pim Nijssen (UM), *M*onte-Carlo Tree Search for Multi-Player Games.
2013-41 Jochem Liem (UVA), *S*upporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning.
2013-42 Léon Planken (TUD), *A*lgorithms for Simple Temporal Reasoning.
2013-43 Marc Bron (UVA), *E*xploration and Contextualization through Interaction and Concepts.

2014

2014-01 Nicola Barile (UU), *S*tudies in Learning Monotone Models from Data.
2014-02 Fiona Tuliyano (RUN), *C*ombining System Dynamics with a Domain Modeling Method.
2014-03 Sergio Raul Duarte Torres (UT), *I*nformation Retrieval for Children: Search Behavior and Solutions.
2014-04 Hanna Jochmann-Mannak (UT), *W*ebsites for children: search strategies and interface design - Three studies on children's search performance and evaluation.
2014-05 Jurriaan van Reijsen (UU), *K*nowledge Perspectives on Advancing Dynamic Capability.
2014-06 Damian Tamburri (VU), *S*upporting Networked Software Development.
2014-07 Arya Adriansyah (TUE), *A*ligning Observed and Modeled Behavior.
2014-08 Samur Araujo (TUD), *D*ata Integration over Distributed and Heterogeneous Data Endpoints.
2014-09 Philip Jackson (UvT), *T*oward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language.
2014-10 Ivan Salvador Razo Zapata (VU), *S*ervice Value Networks.
2014-11 Janneke van der Zwaan (TUD), *A*n Empathic Virtual Buddy for Social Support.
2014-12 Willem van Willigen (VU), *L*ook Ma, No Hands: Aspects of Autonomous Vehicle Control.
2014-13 Arlette van Wissen (VU), *A*gent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains.
2014-14 Yangyang Shi (TUD), *L*anguage Models With Meta-information.
2014-15 Natalya Mogles (VU), *A*gent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare.
2014-16 Krystyna Milian (VU), *S*upporting trial recruitment and design by automatically interpreting eligibility criteria.
2014-17 Kathrin Dentler (VU), *C*omputing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability.
2014-18 Mattijs Ghijsen (VU), *M*ethods and Models for the Design and Study of Dynamic Agent Organizations.
2014-19 Vincius Ramos (TUE), *A*daptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support.
2014-20 Mena Habib (UT), *N*amed Entity Extraction and Disambiguation for Informal Text: The Missing Link.
2014-21 Kassidy Clark (TUD), *N*egotiation and Monitoring in Open Environments.
2014-22 Marieke Peeters (UU), *P*ersonalized Educational Games - Developing agent-supported scenario-based training.
2014-23 Eleftherios Sidirourgos (UvA/CWI), *S*pace Efficient Indexes for the Big Data Era.
2014-24 Davide Ceolin (VU), *T*rusting Semi-structured Web Data.
2014-25 Martijn Lappenschaar (RUN), *N*ew network models for the analysis of disease interaction.
2014-26 Tim Baarslag (TUD), *W*hat to Bid and When to Stop.
2014-27 Rui Jorge Almeida (EUR), *C*onditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty.
2014-28 Anna Chmielowiec (VU), *D*ecentralized k-Clique Matching.
2014-29 Jaap Kabbedijk (UU), *V*ariability in Multi-Tenant Enterprise Software.
2014-30 Peter de Cock (UvT), *A*nticipating Criminal Behaviour.
2014-31 Leo van Moergestel (UU), *A*gent Technology in Agile Multiparallel Manufacturing and Product Support.
2014-32 Naser Ayat (UvA), *O*n Entity Resolution in Probabilistic Data.
2014-33 Tesfa Tegegne (RUN), *S*ervice Discovery in eHealth.

2014-34 Christina Manteli(VU), *T*he Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems..
2014-35 Joost van Ooijen (UU), *C*ognitive Agents in Virtual Worlds: A Middleware Design Approach.
2014-36 Joos Buijs (TUE), *F*lexible Evolutionary Algorithms for Mining Structured Process Models.
2014-37 Maral Dadvar (UT), *E*xperts and Machines United Against Cyberbullying.
2014-38 Danny Plass-Oude Bos (UT), *M*aking brain-computer interfaces better: improving usability through post-processing..
2014-39 Jasmina Maric (UvT), *W*eb Communities, Immigration, and Social Capital.
2014-40 Walter Omona (RUN), *A* Framework for Knowledge Management Using ICT in Higher Education.
2014-41 Frederic Hogenboom (EUR), *A*utomated Detection of Financial Events in News Text.
2014-42 Carsten Eijckhof (CWI/TUD), *C*ontextual Multidimensional Relevance Models.
2014-43 Kevin Vlaanderen (UU), *S*upporting Process Improvement using Method Increments.
2014-44 Paulien Meesters (UvT), *I*ntelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden..
2014-45 Birgit Schmitz (OUN), *M*obile Games for Learning: A Pattern-Based Approach.
2014-46 Ke Tao (TUD), *S*ocial Web Data Analytics: Relevance, Redundancy, Diversity.
2014-47 Shangsong Liang (UVA), *F*usion and Diversification in Information Retrieval.

2015

2015-01 Niels Netten (UvA), *M*achine Learning for Relevance of Information in Crisis Response.
2015-02 Faiza Bukhsh (UvT), *S*mart auditing: Innovative Compliance Checking in Customs Controls.
2015-03 Twan van Laarhoven (RUN), *M*achine learning for network data.
2015-04 Howard Spoelstra (OUN), *C*ollaborations in Open Learning Environments.
2015-05 Christoph Bösch(UT), *C*ryptographically Enforced Search Pattern Hiding.
2015-06 Farideh Heidari (TUD), *B*usiness Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes.
2015-07 Maria-Hendrike Peetz(UvA), *T*ime-Aware Online Reputation Analysis.
2015-08 Jie Jiang (TUD), *O*rganizational Compliance: An agent-based model for designing and evaluating organizational interactions.
2015-09 Randy Klaassen(UT), *H*CI Perspectives on Behavior Change Support Systems.
2015-10 Henry Hermans (OUN), *O*penU: design of an integrated system to support lifelong learning.
2015-11 Yongming Luo (TUE), *D*esigning algorithms for big graph datasets: A study of computing bisimulation and joins.
2015-12 Julie M. Birkholz (VU), *M*odi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks.
2015-13 Giuseppe Procaccianti(VU), *E*nergy-Efficient Software.
2015-14 Bart van Straalen (UT), *A* cognitive approach to modeling bad news conversations.
2015-15 Klaas Andries de Graaf (VU), *O*ntology-based Software Architecture Documentation.
2015-16 Changyun Wei (UT), *C*ognitive Coordination for Cooperative Multi-Robot Teamwork.
2015-17 Andre van Cleeff (UT), *P*hysical and Digital Security Mechanisms: Properties, Combinations and Trade-offs.
2015-18 Holger Pirk (CWI), *W*aste Not, Want Not! - Managing Relational Data in Asymmetric Memories.
2015-19 Bernardo Tabuenca (OUN), *U*biquitous Technology for Lifelong Learners.
2015-20 Lois Vanhée(UU), *U*sing Culture and Values to Support Flexible Coordination.
2015-21 Sibren Fetter (OUN), *U*sing Peer-Support to Expand and Stabilize Online Learning.
2015-22 Zhemin Zhu(UT), *C*o-occurrence Rate Networks.
2015-23 Luit Gazendam (VU), *C*ataloguer Support in Cultural Heritage.
2015-24 Richard Berendsen (UVA), *F*inding People, Papers, and Posts: Vertical Search Algorithms and Evaluation.
2015-25 Steven Woudenberg (UU), *B*ayesian Tools for Early Disease Detection.
2015-26 Alexander Hogenboom (EUR), *S*entiment Analysis of Text Guided by Semantics and Structure.
2015-27 Sandor Héman (CWI), *U*pdating compressed colomn stores.

2015-28 Janet Bagorogoza(TiU), *K*NOWLEDGE MANAGEMENT AND HIGH PERFORMANCE; The Uganda Financial Institutions Model for HPO.
2015-29 Hendrik Baier (UM), *M*onte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains.
2015-30 Kiavash Bahreini(OU), *R*eal-time Multimodal Emotion Recognition in E-Learning.
2015-31 Yakup Ko c (TUD), *O*n the robustness of Power Grids.
2015-32 Jerome Gard(UL), *C*orporate Venture Management in SMEs.
2015-33 Frederik Schadd (TUD), *O*ntology Mapping with Auxiliary Resources.
2015-34 Victor de Graaf(UT), *G*esocial Recommender Systems.
2015-35 Jungxao Xu (TUD), *A*ffective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction.

2016

2016-01 Syed Saiden Abbas (RUN), *R*ecognition of Shapes by Humans and Machines.
2016-02 Michiel Christiaan Meulendijk (UU), *O*ptimizing medication reviews through decision support: prescribing a better pill to swallow.
2016-03 Maya Sappelli (RUN), *K*nowledge Work in Context: User Centered Knowledge Worker Support.
2016-04 Laurens Rietveld (VU), *P*ublishing and Consuming Linked Data.
2016-05 Evgeny Sherkhonov (UVA), *E*xpanded Acyclic Queries: Containment and an Application in Explaining Missing Answers.
2016-06 Michel Wilson (TUD), *R*obust scheduling in an uncertain environment.
2016-07 Jeroen de Man (VU), *M*easuring and modeling negative emotions for virtual training.
2016-08 Matje van de Camp (TiU), *A* Link to the Past: Constructing Historical Social Networks from Unstructured Data.
2016-09 Archana Nottamkandath (VU), *T*rusting Crowdsourced Information on Cultural Artefacts.
2016-10 George Karafotias (VUA), *P*arameter Control for Evolutionary Algorithms.
2016-11 Anne Schuth (UVA), *S*earch Engines that Learn from Their Users.
2016-12 Max Knobbout (UU), *L*ogics for Modelling and Verifying Normative Multi-Agent Systems.
2016-13 Nana Baah Gyan (VU), *T*he Web, Speech Technologies and Rural Development in West Africa - An.
ICT4D Approach **2**016-14 Ravi Khadka (UU), *R*evisiting Legacy Software System Modernization.
2016-15 Steffen Michels (RUN), *H*ybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments.
2016-16 Guangliang Li (UVA), *S*ocially Intelligent Autonomous Agents that Learn from Human Reward.
2016-17 Berend Weel (VU), *T*owards Embodied Evolution of Robot Organisms.
2016-18 Albert Meroño Peñuela, *R*efining Statistical Data on the Web.
2016-19 Julia Efremova (Tu/e), *M*ining Social Structures from Genealogical Data.
2016-20 Daan Odijk (UVA), *C*ontext & Semantics in News & Web Search.
2016-21 Alejandro Moreno Célleri (UT), *F*rom Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground.
2016-22 Grace Lewis (VU), *S*oftware Architecture Strategies for Cyber-Foraging Systems.
2016-23 Fei Cai (UVA), *Q*uery Auto Completion in Information Retrieval.
2016-24 Brend Wanders (UT), *R*epurposing and Probabilistic Integration of Data; An Iterative and data model independent approach.
2016-25 Julia Kiseleva (TU/e), *U*sing Contextual Information to Understand Searching and Browsing Behavior.
2016-26 Dilhan Thilakarathne (VU), *I*n or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains.
2016-27 Wen Li (TUD), *U*nderstanding Geo-spatial Information on Social Media.
2016-28 Mingxin Zhang (TUD), *L*arge-scale Agent-based Social Simulation - A study on epidemic prediction and control.
2016-29 Nicolas Höning (TUD), *P*eak reduction in decentralised electricity systems -Markets and prices

for flexible planning.
2016-30 Ruud Mattheij (UvT), *T*he Eyes Have It.
2016-31 Mohammad Khelghati (UT), *D*eep web content monitoring.
2016-32 Eelco Vriezekolk (UT), *A*ssessing Telecommunication Service Availability Risks for Crisis Organisations.
2016-33 Peter Bloem (UVA), *S*ingle Sample Statistics, exercises in learning from just one example.
2016-34 Dennis Schunselaar (TUE), *T*itle: Configurable Process Trees: Elicitation, Analysis, and Enactment.
2016-35 Zhaochun Ren, *M*onitoring Social Media: Summarization, Classification and Recommendation.
2016-36 Daphne Karreman (UT), *B*eyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies.
2016-37 Giovanni Sileno (UvA), *A*ligning Law and Action - a conceptual and computational inquiry .
2016-38 Andrea Minuto (UT), *M*ATERIALS THAT MATTER - Smart Materials meet Art & Interaction Design.
2016-39 Merijn Bruijnes (UT), *B*elievable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect.
2016-40 Christian Detweiler (TUD), *A*ccounting for Values in Design.
2016-41 Thomas King (TUD), *G*overning Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance.
2016-42 Spyros Martzoukos (UVA) , *C*ombinatorial and Compositional Aspects of Bilingual Aligned Corpora .
2016-43 Saskia Koldijk (RUN), *C*ontext-Aware Support for Stress Self-Management: From Theory to Practice.
2016-44 Thibault Sellam (UVA), *A*utomatic Assistants for Database Exploration.
2016-45 Bram van de Laar (UT), *E*xperiencing Brain-Computer Interface Control.
2016-46 Jorge Gallego Perez (UT), *E*xperiencing Brain-Computer Interface Control.
2016-47 Christina Weber (UL, *R*eal-time foresight - Preparedness for dynamic innovation networks.
2016-48 Tanja Buttler (TUD), *C*ollecting Lessons Learned.
2016-49 Gleb Polevoy (TUD), *T*he title: Participation and Interaction in Projects. A Game-Theoretic Analysis.
2016-50 Yan Wang (UVT), *T*he Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains.
2017-01 Jan-Jaap Oerlemans (UL), *I*nvestigating Cybercrime.
2017-02 Sjoerd Timmer (UU), *D*esigning and Understanding Forensic Bayesian Networks using Argumentation.
2017-03 Daniël Harold Telgen (UU), *G*rid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines.