Program Analysis and Compilation Techniques for Speeding up Transactional Database Workloads

THÈSE N^{O} 8023 (2017)

PRÉSENTÉE LE 13 OCTOBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE THÉORIE ET APPLICATIONS D'ANALYSE DE DONNÉES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Mohammad DASHTI RAHMAT ABADI

acceptée sur proposition du jury:

Prof. E. Bugnion, président du jury
Prof. C. Koch, directeur de thèse
Dr Ph. A. Bernstein, rapporteur
Prof. A. Pavlo, rapporteur
Prof. A. Ailamaki, rapporteuse

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

I have seen flowers come in stony places
And kind things done by men with ugly faces
And the gold cup won by the worst horse at the races,
So I trust, too.
— John Masefield

To my beloved wife, Faeze,
and to my dear son, Daniel.

Acknowledgements

I would like to express my special appreciation and thanks to my advisor, Prof. Christoph Koch. He has taught me, both consciously and unconsciously, how good (database) research is done. I would like to thank him for encouraging my research and for allowing me to grow as a research scientist. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. Christoph taught me how to distinguish between incremental and fundamental advances in science, to think big and to aim for the biggest targets. I am also thankful to him that he almost always pulled an all-nighter with us before a deadline.

I would like to thank Dr. Phil Bernstein, my advisor at Microsoft Research, where I interned during summer 2016. Thank you for providing me with such a great environment and for assembling such a great team of highly qualified and motivated people. It was my pleasure working and brainstorming with you! I learned a lot from Phil, especially in terms of getting great ideas and getting them done in a real system.

Besides my advisor and Dr. Phil Bernstein, I would like to thank the rest of my thesis committee: Prof. Anastasia Ailamaki and Prof. Andrew Pavlo, for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

The members of the DATA Lab have contributed immensely to my personal and professional time at EPFL. The group has been a source of friendships as well as good advice and collaboration. I owe many thanks to Sachin Basil John, my dear colleague. He helped me in so many different ways so that it is hard to express in words all the gratitude I feel towards him. I thank Thierry Coppey for all the fun we had in building software together and in the extracurricular activities. I thank my officemate, Mohammed El Seidy, for the numerous hour-long discussions and brain-storming sessions. Other past and present lab members that I have had the pleasure to work with or alongside are grad students Milos Nikolic, Amir Shaikhha, Aleksandar Vitorovic, Daniel Lupei, Voijin Jovanovic and Yannis Klonatos; and the numerous summer and rotation students who have come through the lab. I also thank my other fellow labmates in DATA Lab for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last five years.

I would also like to thank many of my friends I got to know here in Switzerland. I thank Ashakn, Hesam, and Nooshin for encouraging me during my studies. I am thankful to the Iranian EPFL community for many events we had.

No success is possible without a strong family support. I owe greatest thanks to my beloved wife Faezah for all the love, care, encouragement and understanding. She spent sleepless nights with me and was always my support in the moments when there was no one to answer

my queries. She made my life much more enjoyable. I also thank her for tolerating my long working hours before the deadlines and for patiently listening to the computer science related discussions. I thank our son Daniel, who always ran to me and hugged me when I was coming back from work. He was my biggest source of energy in the past two years. I thank my parents, Mohammadhossein and Malekolsadat, for their unconditional love and moral and financial support throughout my whole life. They taught me what are the core values in life, and to work hard. Your prayer for me was what sustained me thus far.

Mohammad Dashti
EPFL
August 2017

Abstract

There is a trend towards increased specialization of data management software for performance reasons. The improved performance not only leads to a more efficient usage of the underlying hardware and cuts the operation costs of the system, but also is a game-changing competitive advantage for many emerging application domains such as high-frequency algorithmic trading, clickstream analysis, infrastructure monitoring, fraud detection, and online advertising to name a few.

In this thesis, we study the automatic specialization and optimization of database application programs – sequences of queries and updates, augmented with control flow constructs as they appear in database scripts, user-defined functions (UDFs), transactional workloads and triggers in languages such as PL/SQL. We propose to build online transaction processing (OLTP) systems around a modern compiler infrastructure. We show how to build an optimizing compiler for transaction programs using generative programming and state-of-the-art compiler technology, and present techniques for aggressive code inlining, fusion, deforestation, and data structure specialization in the domain of relational transaction programs. We also identify and explore the key optimizations that can be applied in this domain.

In addition, we study the advantage of using program dependency analysis and restructuring to enable the concurrency control algorithms to achieve higher performance. Traditionally, optimistic concurrency control algorithms, such as optimistic Multi-Version Concurrency Control (MVCC), avoid blocking concurrent transactions at the cost of having a validation phase. Upon failure in the validation phase, the transaction is usually aborted and restarted from scratch. The "abort and restart" approach becomes a performance bottleneck for use cases with high contention objects or long running transactions. In addition, restarting from scratch creates a negative feedback loop in the system, because the system incurs additional overhead that may create even more conflicts. However, using the dependency information inside the transaction programs, we propose a novel transaction repair approach for in-memory databases. This low overhead approach summarizes the transaction programs in the form of a dependency graph. The dependency graph also contains the constructs used in the validation phase of the MVCC algorithm. Then, when encountering conflicts among transactions, our mechanism quickly detects the conflict locations in the program and partially re-executes the conflicting transactions. This approach maximizes the reuse of the computations done in the first execution round and increases the transaction processing throughput.

We evaluate the proposed ideas and techniques in the thesis on some popular benchmarks such as TPC-C and modified versions of TPC-H and TPC-E, as well as other micro-benchmarks. We show that applying these techniques leads to 2x-100x performance improvement in many

use cases. Besides, by selectively disabling some of the optimizations in the compiler, we derive a clinical and precise way of obtaining insight into their individual performance contributions.

Key words: transaction processing, program analysis, compilation techniques, transaction repair, generative programming, program optimization, functional programming, compiler, staging, data structure specialization, deforestation, partial evaluation

Résumé

Les logiciels de gestion de données tendent vers une spécialisation accrue, pour des raisons de performance. Cette performance accrue donne lieu non seulement à une utilisation plus efficiente du matériel sous-jacent, réduisant le coût opérationnel du système entier, mais est également un avantage compétitif important, au point de changer la donne pour de nombreux domaines d'application émergents, comme le trading algorithmique à haute fréquence, l'analyse du comportement des utilisateurs, la surveillance des infrastructures, la détection de fraude, ou la publicité en ligne, pour n'en citer que quelques-uns.

Dans cette thèse, nous étudions la spécialisation automatique et l'optimisation de programmes transactionnels - des séquences de requêtes et de mises a jour, augmentées de constructions de contrôle de flot telles qu'elles apparaissent dans les procédures de bases de données, de fonctions définies par l'utilisateur (les UDF), d'opérations gestion de transaction, et de déclencheurs comme ceux d'un langage tel PL/SQL. Nous proposons la construction de systèmes de traitement de transactions en ligne (OLTP) basée sur une infrastructure de compilateur moderne. Nous montrons comment construire un compilateur spécialisé pour l'optimisation de programmes transactionnels, en utilisant la programmation générative et autre techniques de compilation modernes, et présentons des techniques agressives d'expansion de code, de fusion, de déforestation, et de spécialisation des structures de données, dans le contexte des programmes de transactions relationnelles. Nous explorons et identifions également les optimisations clés qui peuvent être appliquées dans ce domaine.

De plus, nous étudions l'intérêt de l'utilisation de l'analyse des dépendances et de la restructuration des programmes, pour améliorer la performance des algorithmes de contrôle de concurrence. Traditionnellement, les algorithmes de contrôle de concurrence optimistes, comme le Multi-Version Concurrency Control (MVCC), préviennent le blocage de transactions concurrentes au prix d'une étape préliminaire de validation. En cas d'échec de l'étape de validation, la transaction est en général annulée, et reprise à zéro. L'approche "annuler et redémarrer" devient le facteur limitant pour la performance, lorsqu'un objet présente une contention élevée, ou dans les transactions de longue durée. De plus, repartir de zéro crée une boucle de rétroaction nocive, puisque le système subit une charge supplémentaire pouvant déclencher des conflits additionnels. Cependant, en exploitant l'information des dépendances présentent dans les programmes transactionnels, nous proposons une nouvelle approche de réparation des transactions pour les bases de données en mémoire. Cette approche à coût réduit représente les transactions sous la forme d'un graphe de dépendances. Ce graphe contient également les constructions utilisées dans l'étape de validation de l'algorithme MVCC. Dès lors, en cas de conflit entre des transactions, notre mécanisme détecte rapidement la position des conflits dans le programme et ré-exécute partiellement les transactions conflictuelles.

Cette approche maximise la réutilisation des calculs effectués dans la phase d'exécution initiale, et augmente le débit de traitement des transactions.

Nous validons les idées et techniques proposées dans cette thèse sur des évaluations standardisées comme TPC-C, et des versions modifiées de TPC-H et TPC-E, ainsi que sur d'autres micro-évaluations. Nous montrons que l'utilisation de ces techniques multiplie la performance d'un facteur 2 à 100 dans de nombreux cas d'utilisation. De surcroît, en désactivant certaines optimisations du compilateur, nous obtenons une manière systématique et précise d'évaluer leurs contributions individuelles à la performance.

Mots clefs : base de données transactionnelles, analyse de programmes, techniques de compilation, réparation des transactions, programmation générative, optimisation de programmes, programmation fonctionnelle, compilation, staging, spécialisation des structures de données, déforestation, évaluation partielle.

Contents

1 Introduction

Software specialization bears great promise for overcoming performance challenges in data management systems. Differently from hardware specialization (cf. database machines such as GAMMA [31]), which is at odds with economies of scale in hardware manufacturing and has been long held to be forbiddingly costly[i], software specialization is relatively cheap. As argued in [108, 109] and elsewhere, it can provide substantial performance benefits. The conclusion obtained in some of this recent work is that we need to design and build special-purpose database systems for different types of workloads (OLTP, OLAP, complex event processing (CEP), and scientific DBMS are a few application classes).

One way to achieve code specialization is by using compiler technology. In particular, just-in-time (JIT) compilers can delay code specialization to runtime and use information about the workload and data to further adapt and optimize the running database system code.

Even database systems tailored for particular workloads can make effective use of compilation to further code specialization. For instance, the case for H-Store and VoltDB was built [109] using heavily optimized hand-written code. The argument was made that, in TPC-C, expert-written code outperforms classical OLTP systems by two orders of magnitude. To make the vision of these systems real, we need to be able to *automatically* produce this super-efficient code from naïve, high-level, and less expertly written transaction programs. This calls for optimizing compiler technology.

The design of the aforementioned novel OLTP systems assumes that users and applications interact with the server via stored procedures representing transaction code: the transaction code is moved to the server. The logical conclusion is to have a compiler infrastructure at the heart of such a system which allows for the necessary code optimization and the dynamic loading of freshly compiled code during the operation of the system. Compiler ecosystems such as LLVM [72] make this possible; even more recent systems such as SC [103] and LMS [98] make the development of advanced optimizations and data/storage structure specialization relatively easy.

[i] We note that database machines are currently experiencing a renaissance nevertheless, cf. the Oracle Exadata appliance and project RAPID in Oracle Labs, which was abandoned in 2017.

For forty years, since the earliest times of relational DBMS, we have studied the compilation of query plans [103].[ii] Recently, we have seen a revival of query compilation in contexts such as stream processing systems (IBM Spade and Streambase) and analytical queries (DBToaster [1], LegoBase [64] and Hyper [83]). However, the optimized compilation of transaction code is largely unknown territory, only recently entered by MemSQL [104] and Microsoft's Hekaton system [32], for a limited version of TSQL –their domain-specific language (DSL) for transaction programs. Good speedups compared to classical interpreted evaluation have been reported, but it is still open whether such a compiler can rival the performance hand-optimized code can achieve (cf. [109]).

In addition to the optimizations that are feasible to be applied directly to the database applications and transaction programs, other components of a transaction processing system can also benefit from analyzing the dependencies among operations of the programs. One of the most performance critical components in a transaction processing system is its concurrency control module. However, the standard concurrency control algorithms used in practice do not take advantage of the information encoded inside the transaction programs and only rely on the effect of their issued operations, i.e., read and write commands on data objects.

1.1 Thesis statement

In this thesis, we study the compilation of transaction programs and using program analysis to improve concurrency control algorithms. We propose how to build such a compiler-based infrastructure for whole-system specialization. To us, this is the logical completion of the vision of H-Store and VoltDB. Furthermore, it is also applicable to a wide range of other scenarios, such as specialization of otherwise more classical OLTP systems (e.g., Hekaton [32]), and the compilation of incremental view maintenance trigger programs in systems like DBToaster [1].

In contrast to queries, which are phrased in a limited language, transaction programs have much more variability and combine queries, updates, and general-purpose code. A compiler's internal code representations require much greater generality than those of a query compiler. But once such a powerful language is supported, the system can in principle compile, inline and optimize the transaction programs *together* with generic server code, achieving a maximal degree of specialization and thus performance. Compiler technology has never been used in such an ambitious way before in our community, although a case can be made that the time is ripe to attempt this [65].

Compiling OLTP transaction code opens up the potential for important optimizations that are feasible in this domain, but which no current compiler supports, and which are in general impossible or meaningless outside of the domain of transaction programs. The main reason for this is that transaction programs tend to use the database's storage structures (tables, indexes, or the database connectivity library's proxies – "result sets") as their only dynamic data

[ii] In the very beginning of the System R project, query execution was based on compilation. Because the seventies compiler technology was inconvenient in a rapidly progressing project, this compiler was replaced by the now standard interpretation approach just before the first release. Also many new ideas had to be explored: it was much faster to implement an algorithm directly than to write a second algorithm that outputs the code of the first [17].

structures. Creating special compiler optimizations related to such *collection* data structures can thus be highly useful, particularly if one keeps in mind that domain-specific (abstract) data types are a natural focal point for most domain-specific compiler optimizations.

Moreover, as transaction programs are usually running concurrently and in parallel, the choice of concurrency control mechanism in the execution engine of the OLTP system plays a significant role in its performance. In this thesis, we also study the impact of using different concurrency control techniques on overall transaction processing performance in the presence of using program analysis and compilation techniques. We look into this matter from two perspectives: 1) we show the impact of the choice of concurrency control model on the effectiveness of transaction programs that are optimized using our compilation suite, and 2) we look at possible ways of improving the concurrency control algorithms by taking advantage of the program analysis and restructuring.

The concurrency model used in H-Store [54] is the best match for applying aggressive optimizations on the transaction programs, as all the transactions on a partition are sequentially handled by an isolated worker thread, without any data sharing with other threads. This assumption gives more flexibility to the transaction compiler to apply the optimizations that are only correct in the case of having a single thread that accesses and manipulates the underlying data structures.

Even though the H-Store concurrency model performs best for partitionable transactional workloads [54], not all workloads are perfectly partitionable. The concurrency control algorithms that operate on a shared memory model (e.g., multi-version concurrency control (MVCC), optimistic concurrency control (OCC), two phase locking (2PL)) might be a better fit for the latter workloads. Even though some optimizations are not applicable when these concurrency control algorithms are used, we show that using program analysis and compilation techniques still has a significant impact on the performance improvement of transaction programs.

Interestingly, the concurrency control algorithms that operate on a shared memory model have the potential of benefiting from dependency information inside the transaction programs. This dependency information combined with program analysis can be used to either: 1) restructure the programs in a way that improves the concurrency control performance, e.g., detecting conflicts between transactions at the beginning of the program instead of its end in optimistic concurrency control algorithms, or 2) help the internal operations of the concurrency control algorithm to be more efficient. Given the dynamic and runtime nature of the concurrency control algorithms, it seems that program analysis cannot do much when it comes to the internals of these algorithms, as they prevent a conflict between operations in different instances of transaction programs at runtime. However, using dependency information among operations inside concurrency control algorithms, we propose a new concurrency control algorithm, called Multi-Version Concurrency Control with Closures (MV3C), which has an approach for effective conflict resolution between transactions.

When a conflict happens between transactions, instead of aborting and restarting them from scratch, MV3C partially repairs the conflict. The main challenges for this type of conflict resolution technique are having: (1) low overhead on the standard execution of transactions, as every transaction pays this cost irrespective of encountering a conflict, and (2) a fast

mechanism to narrow down the conflicting portions of the transactions and fixing them, as a mechanism that is slower than the abort and restart approach defeats the purpose. MV3C deals with the first challenge by reducing the additional bookkeeping by reusing the existing concurrency control machinery. To address the second issue, MV3C uses lightweight dependency declaration constructs in the transaction programs that help in immediately identifying the dependencies among different operations. Using this dependency information, MV3C pinpoints blocks of the program affected by the conflicts and quickly re-executes only those blocks. The dependency information is added either manually by the user, or by employing static program analysis and restructuring.

The rationale for proposing MV3C is that a conflict happens only if a transaction reads some data objects from the database that become stale by the time it tries to commit. Here, the assumption is that all data modifications made by a transaction are invisible to other transactions during its execution. These changes become visible only after the critical section during which the transaction commits. Consequently, just before committing a transaction, by checking whether its data lookup operations read the most recent (committed) versions of the data objects, serializability of the execution is ensured. In addition, by associating read operations with the blocks of code that depend on the read data, the portion of a transaction that should be re-executed in the case of a conflict is identified quickly and gets re-executed.

The transaction repair approach proposed in MV3C is most useful for workloads with either: (1) high contention data objects that are read and updated by several concurrent transactions, or (2) long running transactions, the lifespan of which intersects that of many other transactions. For the former type of workloads, MV3C reduces the contention by minimizing the contention timespan, as the repair phase is usually much shorter than a full "abort and restart." For the latter type of workloads, MV3C prevents starvation of the long-running transactions, by giving them a better chance in competing with short-running transactions, as their repair phase is potentially much shorter. This problem has been a long-standing issue with the optimistic multi-version concurrency control algorithms that longer transactions could have faced starvation in the race with shorter transactions.

1.2 Contributions

This thesis studies how to use program analysis and compilation techniques for efficiently compiling transaction programs in conjunction with the underlying database and system code. This work includes material from several publications in which author is the lead author or a co-author. The contributions brought forth in this thesis are the following:

- A transaction execution infrastructure with a staging compiler in its heart. The staging allows flexibility in the database application implementation (interpretation or compilation)[iii]. The architecture of this infrastructure is explained in section 3.2.
- A high-level *domain-specific language (DSL) for transaction programs* that facilitates domain-specific optimizations which result in higher performance. The semantics of this DSL is explained in section 3.3.

[iii] The user can tune inlining of the database internal functions in, thereby trading efficiency for compilation time.

- Compiler optimizations for the domain of transaction programs and our DSL, in particular automatic data structure specialization (simplifying the internals of data structures where possible) and deforestation (optimizing across operators). These optimizations are described in section 3.4.
- The automatic choice and specialization of storage and index structures (array, hash, tree, heap) based on transactional workload patterns, and the automatic adaptation of application code to use these data structures. This specialization and indexing technique is explained in sections 3.4.2 and 3.5.
- The breakdown of the impact of each optimization on standard benchmarks (e.g. TPC-C and TPC-H) as well as some further micro-benchmarks. These results are presented in section 3.6.
- An efficient conflict resolution mechanism for multi-version databases, MV3C, which repairs conflicts instead of aborting transactions, with minimal execution overhead. The design of this mechanism is discussed in section 4.2.
- A method to deal with write-write conflicts in MV3C. Unlike other optimistic MVCC algorithms, MV3C can avoid aborting a transaction prematurely when a write-write conflict is detected. This method is discussed further in section 4.2.3.
- A mechanism for fixing the result-set of failed queries for MV3C, which can optionally be enabled for each query in a transaction. This approach boosts the repair process for transactions that have long running queries. The details of this mechanism are described in section 4.4.2.

The central part of the thesis composes of the following two manuscripts:

- *Mohammad Dashti, Sachin Basil John, Thierry Coppey, Amir Shaikhha, Vojin Jovanovic, and Christoph Koch.*
  Compiling Database Application Programs.
  Under submission.

- *Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch.*
  Transaction Repair for Multi-Version Concurrency Control.
  SIGMOD 2017.

The above papers are presented in chapters 3 and 4, respectively. The first manuscript is under submission. In both of them, the author of this thesis is the lead author and has written and edited the majority of the text. He is the main contributor of the ideas and their development. He also implemented the first prototypes to prove the concepts. He was also the leading contributor to the final implementation of these prototypes for doing the evaluations, the primary contributor to designing the experiments, and the primary contributor creating the setup to experiment with other systems or algorithms. Sachin Basil John contributed to the implementation of the system prototypes and algorithms behind these manuscripts as part of his master thesis research. Thierry Coppey contributed to the development of the ideas and the implementation of an early version of the final prototype for the first manuscript. Vojin Jovanovic helped by revising the first manuscript. Amir Shaikhha contributed by reviewing both manuscripts and also assisted in the development of the idea in the second manuscript.

The material presented in this thesis is indirectly related to the contributions of the thesis author to the other research projects, some of which has already been published. In the following, these published papers are listed, and their connection with this thesis is briefly explained.

- The most important building block for creating a compilation suite to optimize transaction programs has an optimizing compiler in its core. The author of this thesis contributed to two modern optimizing compiler efforts, LMS [98] and SC [103].
  - *Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch.*
    How to Architect a Query Compiler.
    SIGMOD 2016.
  - *Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, and Christoph Koch.*
    Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems.
    SNAPL 2015.
  - *Tiark Rompf, Nada Amin, Thierry Coppey, Mohammad Dashti, Manohar Jonnalagedda, Yannis Klonatos, and Martin Odersky.*
    Abstraction Without Regret for Efficient Data Processing.
    Data-Centric Programming Workshop 2014.
- The ideas and techniques for using program analysis introduced in this thesis are mainly built around generating efficient programs that run in a single machine. However, in the following publications, the author contributed towards creating efficient distributed data processing engines for multiple application domains.
  - *Milos Nikolic, Mohammad Dashti, and Christoph Koch.*
    How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates.
    SIGMOD 2016.
  - *Philip A. Bernstein, Mohammad Dashti, Tim Kiefer, and David Maier.*
    Indexing in an Actor-Oriented Database.
    CIDR 2017.
  - *Aleksandar Vitorovic, Mohammed Elseidy, Khayyam Guliyev, Khue Vu Minh, Daniel Espino, Mohammad Dashti, Ioannis Klonatos, Christoph Koch.*
    Squall: Scalable Real-time Analytics.
    VLDB Demo 2016.

  These systems benefit from extending the ideas presented in this thesis, as well as using efficiently compiled (transaction) programs in each node of the distributed system. Even after optimizing communication cost, which is usually the dominant factor in determining the performance of a distributed system, having efficient information processing routines in

each individual node has the most impact. That is why the ideas and techniques presented in this thesis are also relevant for distributed data processing systems.

1.3 Thesis Outline

Chapter 2 gives an overview of the standard methods used for program analysis and compilation and the recent advances in this area. Many of these techniques are used and extended in the body of the following chapters. Moreover, it covers the concepts from the transaction processing community that are relevant to this thesis. The common concurrency control models and mechanisms and their internals are explained in this chapter.

The first part of the thesis, presented in chapter 3, revolves around using program analysis and compilation techniques for efficiently compiling transaction programs in conjunction with the underlying database and system code. Then, chapter 4 builds upon the idea of using program dependency analysis to enhance the concurrency control algorithms. The concrete approach presented in this chapter proposes an efficient transaction repair mechanism for the optimistic MVCC.

Chapter 5 covers the other works that are related to this thesis and specifies their relation to the ideas and techniques presented in this thesis. Finally, the conclusion of the thesis is provided in chapter 5 and also proposes some future work directions.

2 Background

This chapter serves as a brief overview of relevant subjects in the programming languages and transaction processing communities that are relevant to this thesis.

2.1 Transaction Processing

Transaction Processing is a category of information processing in which the units of work are indivisible operations called *transactions*. The concept of transactions has roots in the context of database management systems as a paradigm for providing concurrent accesses to a shared database and for having an approach to handle failures safely. A transactional database server satisfies the following properties for each transaction, also known as *ACID properties* [50]:

- **Atomicity:** When a transaction reaches its end, if it is successful, then all of its effects are visible to the other transactions. Otherwise, none of its effects should become visible to the other transactions.
- **Consistency preservation:** A transaction starts from a consistent state of the database and its execution transforms the database into a new consistent state.
- **Isolation:** A transaction is isolated from other transactions and executes as if it were the only transaction running on the database and has all the resources to itself.
- **Durability:** When a transaction declares a successful termination, its effects on the database are guaranteed to survive subsequent software or hardware failures.

The primary requirement for a transaction processing system is to provide the ACID properties to the transactions issued by an application program. This requires the system to provide at least the following two core modules [127]:

- **Concurrency control component:** to provide the isolation property between the transactions that run concurrently or in parallel. The protocols and algorithms used in this component are further discussed in section 2.3.
- **Recovery component:** to provide the atomicity and durability properties of transactions.

The consistency preservation property may or may not be explicitly supported by the transaction processing system. One category of systems have the assumption that the transaction

itself should be consistency preserving and do all the required checks during its execution, and the transaction processing system supports a failure in consistency checks by providing the *rollback* feature. The other category of systems has logical rules to enforce the consistency checks and apply them after the successful execution of each transaction.

In addition to providing the ACID guarantees, a transaction processing system should also meet some non-functional requirements, which can be summarized as *providing good performance* [127]. The performance is measured based on either a fixed software/hardware configuration or a performance per unit of cost when the configuration is not fixed. In this context, good performance usually refers to the following:

- **High throughput:** the number of transactions that are successfully processed per unit of time.
- **Low latency:** the duration between issuing a transaction request from a client and receiving a successful end of execution message back to the client.

In fact, if performance were not an issue, providing ACID properties would be very simple. However, to achieve a good performance, a transaction processing system should execute transactions concurrently and in parallel (if the underlying hardware has multiple processing cores). In the next section, we explore the conventional architectures proposed for the parallel transaction processing.

2.2 Parallel Transaction Processing Architecture

There are two mainstream approaches for the implementation of parallelism inside databases and transaction processing systems: shared-everything and shared-nothing. These two schemes differ in whether multiple processors work on a shared physical data layout or not, respectively. The shared-everything architecture is further categorized into shared-memory and shared-disk approaches, which determines whether the location of shared data is in main memory or on disk, respectively. A schematic of these architectures is shown in Figure 2.1, and they are further explained in the following subsections.

2.2.1 Shared-Nothing

In the shared-nothing architecture, at least conceptually, each processor has its memory and disk space that is not shared with another processor. Then, if required, each processor communicates with other ones through passing messages. Thus, if a transaction needs to access a piece of data, it contacts a CPU core that owns this subset of data. In general, the systems that have this architecture are also known as Massively Parallel Processing (MPP) systems.

H-Store [109] is one of the most famous research efforts on using the shared-nothing architecture for transaction processing. Besides H-Store, most other high performance and scalable database management systems tailored for OLAP workloads such as Greenplum, Vertica, DB2, Teradata, and Paraccel follow this architecture. This is also used by the well-known Internet giants to scale their platforms, including Google, Amazon, Facebook, Yahoo, and Akamai.

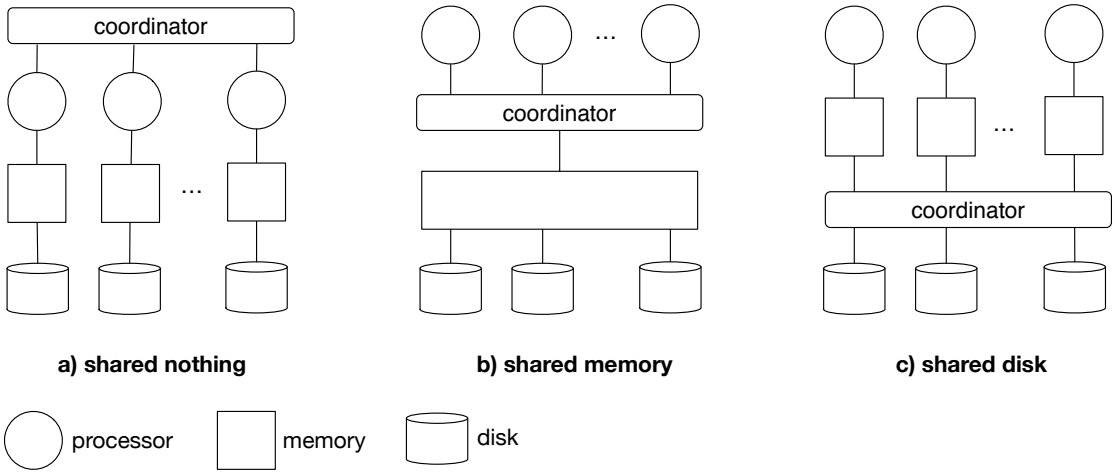


Figure 2.1 – The variants of parallel transaction processing architectures

The assumption in the shared-nothing architecture is that the data is partitioned among the available processors. The partitioning scheme has a significant role in the performance of the system [36, 102]. If the data is partitioned in a way that every request can be handled using a single partition, then no inter-partition communication is required. This is the scenario on which H-Store (and its commercial version, VoltDB) built their case. In this scenario, each processor has a queue of outstanding transactions for its corresponding partition and handles them one-at-a-time in serial order. Therefore, the ACID properties are easily achieved, without any overhead of parallel processing, and in theory it can reach a linear scale-up and scale-out.

However, there are some issues associated with creating a transaction processing system on a shared-nothing model:

- **Distributed transactions:** If there are transactions in the workload that span multiple partitions, then they have to synchronize their operations over the network. Usually, this leads to a considerable slowdown and poor performance.
- **Load imbalances:** There is no guarantee that the load is balanced across all partitions, and it is common to have data skew in partitions. The consequence of a data skew is having more load on the skewed partitions, while other partitions are idle.
- **High repartitioning cost:** After making the initial decision for the partitioning scheme, if one wants to change it, then it usually requires halting the operations of the systems until the repartitioning is done. This can be a very time-consuming task if the underlying database is massive.

The H-Store/VoltDB team have some solutions for these issues. For handling the distributed transactions, they compare two different approaches [54]: 1) using a locking scheme, and 2) using a type of speculative concurrency control. They show that the speculative concurrency control outperforms the locking approach if the number of transaction aborts is not high. For the high repartitioning cost and load imbalances, they propose Squall [36] and Clay [102] that are adaptive partitioning schemes.

2.2.2 Shared-Everything

In this architecture, either memory or disk are shared among multiple processing units. The shared-disk model was first used in VAXcluster [67] and became popular in the 1990's under the name "disk cluster." Sun Microsystems (which was later acquired by Oracle) and Hewlett-Packard (HP) were the primary drivers behind this movement.

The shared-disk model has scalability issues when it is applied to distributed database systems. The main reason is that each processor requires its own lock table and buffer pool, and these resources must be synchronized with the other processors. This synchronization is the key performance bottleneck, which limits the scalability of shared-disk implementations [121].

The shared-memory architecture is a well studied concept in database systems and many concurrency control algorithms are proposed for this model. As multiple processes access the same data in memory, a mechanism should exist to avoid data corruption and guarantee the ACID properties. We will talk more about algorithms to achieve this goal in section 2.3.

The bright side is that, using the shared-everything architecture, there is no need to have a pre-defined partitioning scheme to achieve parallelism. Almost all operations are parallelized without any assumption about the underlying data access pattern. Moreover, if the partitions are coarse-grained, then a shared-everything model can also be embedded inside a shared-nothing architecture.

2.3 Concurrency Control Algorithms

Concurrency control is the problem of applying synchronization among concurrent transactions (i.e., putting an order to the operations done by the concurrent transactions) while the isolation property (in ACID) is maintained. The other important goal is to achieve the maximum performance (i.e., high throughput and low latency) using the available resources [127].

To guarantee the isolation property, we should define the acceptable behavior. One such acceptable behavior is to run the transactions in a serial order. However, it is not necessary to achieve a predefined serial order and only if we had an algorithm that could make sure that transactions ran in one serial order among all possible serial ordering of transactions, then it is also an acceptable behavior. This is implied by the *Serializability* theory. There are several notions of correctness in the literature, e.g., Final-State Serializability, Conflict Serializability, View Serializability, and Commit Serializability to name a few. More details and formal definitions are discussed in the literature [7, 127].

Among all the correctness notions, the most relaxed one that can be used in practical DBMS implementations is *Commit Serializability* [88, 127], which guarantees that any prefix of the interleaving of concurrent operations in transactions is also serializable. The reason for the practical importance of Commit Serializability is that the real-life applications are not usually running in a failure-free environment, and there is no guarantee that each transaction always reaches a successful termination point. As each transaction might fail at any point, the concurrency control algorithm makes sure that all the committed transactions until this point

are serializable. Therefore, as the transaction recovery module guarantees the durability (in ACID) of committed transactions, it is ensured that the isolation property is not violated in the presence of any failure.

Beside the correctness notions, the concurrency control algorithms proposed in the literature can also be categorized in several other ways. One way to categorize these algorithms is based on the number of versions they maintain for each data object, which is either single-version or multi-version. In the rest of this section, we will focus on single-version concurrency control algorithms. Multi-version concurrency control is explained in section 2.3.1.

Another way to categorize concurrency control algorithms is based on whether they use locks in their implementation or not. The main idea in locking algorithms is to synchronize access to shared data by using locks. Locking algorithms set and remove these locks from data objects on behalf of the transactions. Intuitively, while a lock is held by a transaction, other concurrent transactions are blocked from setting an incompatible lock on the same data item until that lock is released. The most well-known locking algorithms are two-phase locking (2PL) and its variants. A locking concurrency control is two phase if, for every transaction, we can distinguish a first phase when locks are only acquired, followed by a second phase when locks are only released. 2PL is not deadlock-free and requires either a deadlock detection or deadlock prevention algorithm [7, 127].

If one wants to get rid of locks, the first approach is using timestamps. The timestamp in this context does not refer to the actual wall-clock time, but it can be a sequence of values from a totally ordered domain. It is usually assumed that a sequence of natural numbers is used as timestamps. *Timestamp Ordering* [7, 127] is an algorithm that uses timestamps. In this algorithm, each transaction t_i is assigned a timestamp $ts(t_i)$ from a virtual clock when it starts. Then, the conflicting operations are ordered based on the timestamp of their transactions, i.e., if two operations have conflict, then the one from the transaction with a lower timestamp must be executed before the one from the transaction with a higher timestamp.

Another non-locking concurrency control algorithm is *Serialization Graph Testing* (SGT) [15]. In this algorithm, each transaction is represented by a vertex and a directed edge is inserted from vertex v_1 to v_2 for each operation $o_1(x)$ (that belongs to v_1) that executes before and is in conflict with an operation $q_2(x)$ (that belongs to v_2). After adding each edge, the algorithm checks whether the graph is still acyclic. If a cycle is found, the transaction that issued the operation is either blocked or restarted [7, 127].

All the algorithms mentioned above (either locking or non-locking) are considered pessimistic algorithms (also known as "conservative"), because they interrupt and block the execution of a transaction during its execution, to make sure that the correctness notion is not violated. There is another category of concurrency control algorithms that are "Optimistic". In optimistic concurrency control, each transaction has three stages. The first stage is the "Read phase" during which the operations of the transaction are executed, but all of its modifications to data are stored in a private workspace that is not visible from other transactions. After the successful execution of the transaction, it goes to the second phase called the "Validation" phase. This phase checks whether the execution violated correctness. If it did, the transaction is aborted and restarted from scratch. Otherwise, it goes to the last stage, the "Write phase," where it writes the contents of its private workspace into the shared database [69, 7, 127].

2.3.1 Multiversion Concurrency Control

As mentioned earlier, there is a broad category of concurrency control algorithms that uses multiple versions of the underlying data objects, instead of having exactly one copy each. As a result, write operations create a new version of each updated data object instead of overwriting it[i] (which is also known as "copy-on-write"). This makes it possible to access the older versions of a data object that is subject to overwriting, at least until the transaction that is writing the new version commits.

There are multiple algorithms in this category. Here, we present two of them: *multi-version timestamp ordering* (MVTO) [93, 94] and *optimistic multi-version concurrency control* (OMVCC) [84]. MVTO has been used in commercial and open-source database systems (e.g., Oracle and Postgres) for decades. OMVCC is more recently proposed in 2015 and has the potential of becoming the mainstream approach.

MVTO transforms operations on data objects into operations on versions of data objects and performs them in a first-in-first-out (FIFO) fashion. These operations are processed in a way that the result appears as if it was produced by a single-version algorithm with transactions in the order of their assigned timestamp. Each transaction t_i is assigned a timestamp $TS(T_i)$ at the beginning of its execution. Then, here are the steps taken by MVTO [127]:

- A read operation $R_i(x)$ is transformed into an $R_i(x_k)$ operation, where x_k is the version of data object x (written by transaction T_k) with largest timestamp less than or equals to $TS(T_i)$.
- A write operation $W_i(x)$ is performed as following:
  - If read operation $R_j(x_k)$ already accounted, where $TS(T_k) < TS(T_i) < TS(T_j)$, then the algorithm rejects $W_i(x)$ and T_i is aborted.
  - Otherwise, $W_i(x)$ is transformed into $W_i(x_i)$, which creates a new version x_i of data object x with $TS(T_i)$ as its timestamp.
- The commit operation C_i for transaction T_i is blocked until all other transactions T_j that have written new versions of data items read by T_i successfully commit. If any of them abort, then T_i must abort and restart from scratch because it read uncommitted data that is not valid anymore.

OMVCC, similar to MVTO, does not block transactions during their execution. While executing a transaction, it gathers the predicates for all the read operations inside the transaction. A predicate in OMVCC can be thought of as a logical predicate created using the attributes of a relation, encapsulating a data selection criterion. For example, the WHERE clause in a SELECT statement on a single table is a predicate for that table. Moreover, OMVCC makes a reasonable assumption that every transaction writes into only a limited number of data objects. Therefore, it is feasible to keep track of the write-set of a transaction (i.e., the *undo buffer* in OMVCC) during its execution. While updating the value of a data object, a new version is created for it by the transaction.

[i] Some variants of MVCC limit the number of versions, e.g., the implementation of Multi-Version concurrency control with Two-Phase Locking (MV2PL) in MySQL and Oracle.

In OMVCC, a **version** is a quadruple (T, O, A, N), where T is the commit timestamp of the transaction that created the version, or the transaction ID if the transaction is not committed, O is the identifier of the associated data object, A is the value of O maintained in this version, and N is a version identifier if more than one version of O is written by T.

Each data object keeps a list of versions belonging to it, called its *version chain*. When a version is created for a data object, it is added to the head of this chain. As an optimization for data storage and retrieval, the value in the new version is written directly into the data object itself, and the old value is stored in the newly created version. This version storage technique is used in OMVCC to improve the scan performance, as it avoids some pointer chasing. The definition of version in the previous paragraph still holds, even with this storage optimization, though the value for each version is not physically stored in the version itself, in a fixed and deterministic offset from it. If a version V is in the head of the version chain, its actual value is in the data object itself. Otherwise, the newer version of V is holding the actual value that belongs to V.

The value written in a version is immutable and cannot be modified, even by its owner transaction. Thus, given a version V, the version identifier N in V is used to differentiate distinct versions written by the same transaction for the same data object. However, after a transaction commits, only the newest version written by the transaction becomes visible to the other transactions. In practice, N can be a pair of pointers, pointing to the older and newer versions in the internal chain of versions written by a single transaction. Then, the committed version is the one without a newer version in N. More formally, a **committed version** is a triple (T, O, A), where (T, O, A, N) is a version such that N is the identifier of the latest version of O written by the transaction with commit timestamp T.

A read operation in OMVCC returns the value inside the data object itself if the version chain is empty. Otherwise, the visible value is reconstructed from the value inside the data object by applying the changes from the version chain, until a visible version is reached. A visible version is either owned by the transaction itself or the latest committed version before the transaction started. In other words, a version (T_1, O, A, N_1) is visible to a transaction with start timestamp T_2 if either:

- T_1 is committed, $T_1 < T_2$ and there is no other committed version $(T_3, O, \_, \_)$ where $T_1 < T_3 < T_2$, or
- $T_1 = T_2$ and there is no other version $(T_1, O, \_, N_2)$ where N_2 is newer than N_1.

OMVCC transactions have an undo buffer that maintains the list of versions created by the transaction. When a transaction commits, its undo buffer contains only the committed versions. As the following proposition suggests, the undo buffer is a representative of the effects of a committed transaction and can also be reused for transaction recovery.

In OMVCC, the only effects of a committed transaction that are visible to other transactions are the committed versions in its undo buffer. Moreover, to capture the changes done to the database between two timestamps, it is sufficient to look at all the transactions that committed during this time period, as the following corollary shows. In other words, the changes to a multi-version database between two timestamps S and C are represented by the versions in the undo buffer of the transactions committed between S and C.

As OMVCC is an optimistic algorithm, it requires a validation phase before a successful commit. A variant of *precision locking* [57] is used for validating OMVCC transactions. This approach requires that the result-sets for all the read operations are still valid at commit attempt time, as if the operations were done at that time.

In order to achieve this, all committed transactions are stored in a list called *recently committed* transactions. During the validation of a transaction, all of its predicates are checked against all committed versions of the transactions in the recently committed list. From this list, only those transactions that committed during the lifetime of the one being validated are considered. The undo buffers of these transactions contain the changes during this time period. If any committed version satisfies a predicate, then the data read by the transaction is obsolete. The newer version should have been read instead, if the read operation used the commit timestamp as its reference point. In this case, the transaction fails validation, and is rolled back and restarted.

2.4 Query Processing Architecture

Most database systems translate a query into an expression in an algebra [27], and then evaluate this expression for producing query results [83]. This evaluation is traditionally done using an iterator model, which is also known as Volcano Style Processing [43]. In this model, each operator receives a tuple stream from each of its inputs, and for iterating over one of them, it only calls the *next* operator of the stream to obtain the successor element.

The iterator model, despite its simplicity, flexibility, and popularity, is not suitable for main memory databases [83], for three reasons. First, it is required to call *next* for every single tuple. Second, invoking *next* is usually a call via a function pointer or a virtual call, which compared to a regular invocation is much more expensive and decreases the precision of branch prediction in modern CPUs. Third, this model often results in bad data and bad instruction cache locality and frequent instruction mis-prediction, and it requires complex state-memorization for storing the current state of execution.

Modern systems change this model by introducing block processing features for receiving more than one tuple in each call to *next* [8, 86]. Amortizing the cost of calling another operator as well as the ability to perform vectorized operations can be listed as advantages of this method. Its worst disadvantage is its inability to pipeline. Pipelining means that an operator passes data to its parent operator without copying or otherwise materializing the data [83]. The block processing and vectorized approach is used in VectorWise [136]. Some other systems such as H-Store [58] and MonetDB [9] use the extreme form of it by materializing the whole intermediate results.

The algebraic operator model [27] is good for reasoning over the query and applying several optimizations. However, it might not be suitable for query processing and producing final results. For more efficient query evaluation, another viewpoint is using the query compilation strategy [64, 83, 103]. In this approach, the algebraic operator model is still used for the first part related to reasoning about the query, but it differs from that model in three ways: First, it is possible to do data-centric query processing, rather than an operator-centric approach. Second, data is pushed toward operators, except being pulled from operators. Third, queries

are compiled and optimized further into native machine code, rather than being interpreted each time.

Moreover, using a compilation strategy for query processing produces executables that rival hand-coded query execution plans and even in some cases, outperform hand-written code. In addition, we automatically benefit from future compiler optimizations and hardware optimizations, almost for free [64].

A high level architecture for query processing is shown in Figure 2.2. This architecture is conceptually derived from [83]. Its main purpose is to reflect the idea in [83] about the best place for putting the query compiler module. It argues that instead of directly compiling a SQL query, we should leverage the benefits of possible optimizations in an algebraic model. The output of this step is used as the input of the query compiler module, which generates the final query evaluator engine.

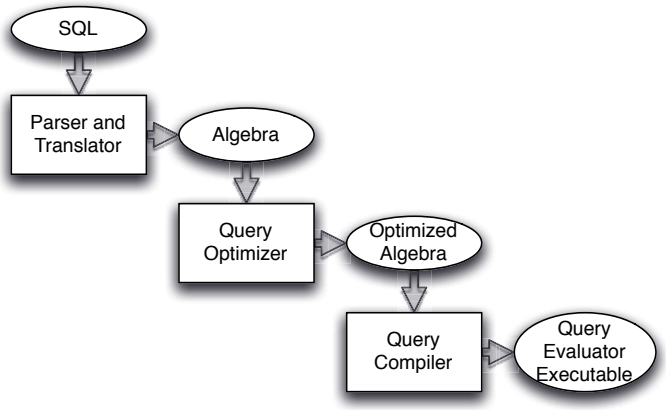


Figure 2.2 – A high-level query processing architecture using a compilation strategy (proposed in [83])

2.5 Query Compilation

There are several previous efforts targeting query compilation. In [92] the authors proposed a method for compiling query logic into Java Bytecode which allows for using the *Java Virtual Machine (JVM)*. However, this is relatively heavy weight [83] and they still use iterator model in their approach which limits the benefits.

In another work, the HIQUE system [66] and MemSQL [104] suggested query compilation of C code using code templates for operators. By inlining the result of materialization inside the operator execution, HIQUE removes the iterator model. However, compared to the approach used in [83], operator boundaries are still visible and there is a high cost of compiling generated C code.

In [83] the author proposes a new concept, named *pipeline-breaker*. A pipeline-breaker is an algebraic operator that for a given input side, takes an incoming tuple out of CPU registers. The author himself admits the definition is slightly hand-waving, because a single tuple might

already be larger than available CPU registers, but he assumes that we have enough registers. The main point about a pipeline-breaking operation is the fact that we need to spill data into memory when we want to apply this operation.

Moreover, it is proposed in [83] to reverse the direction of data flow control, so that the data stream is pushed to operators, rather than being pulled, as in Volcano Style query processing models. With this assumption, we continue pushing data until we reach a pipeline-breaker. As a result, data is always pushed from one pipeline-breaker to another pipeline-breaker. There are two advantages in this method: First, enclosed operators do not manipulate the tuples in the CPU registers and they become very cheap operations. Second, complicated control flow logic remains outside of tight loops and reduces the register pressure.

In addition, instead of the simple common interface in the iterator model that each operator only implements a *next()* function, [83] introduces a new interface consisting of two functions, *produce()* and *consume(attributes, source)*. Such an abstraction helps to keep the code maintainable, even though in this case, it only exists in the query compiler and is only a mental model. A sample translation scheme to illustrate the produce/consume interaction for generating pseudo code is shown in Figure 2.3. In practice, the real translation code is more complex, because it keeps track of the loaded attributes, the operators involved in the evaluation, and the attribute dependencies if an attribute is referenced by a subquery.

⋈.produce	⋈.left.produce; ⋈.right.produce;

⋈.consume(a,s)	if (s==⋈.left)
	print "materialize tuple in hashtable";
	else
	print "for each match in hashtable["
	+a.joinattr+"]";
	⋈.parent.consume(a+new attributes)

σ.produce	σ.input.produce
σ.consume(a,s)	print "if "+σ.condition; σ.parent.consume(attr,σ)
scan.produce	print "for each tuple in relation" scan.parent.consume(attributes,scan)

Figure 2.3 – A sample translation scheme for showing produce/consume interaction (proposed in [83])

The output language of a query compiler is also an important issue. There is a possibility of generating code in a higher level language like C++, which easily connects to other existing systems, But in this case, one will lack total control over the generated code (e.g for accessing overflow flags). There is also a discussion in [83] about the compilation time needed for an optimizing C++ compiler to produce an executable, which is a relevant feature in this context. But it might not be a critical feature in some applications that involve using a query for a long period.

On the other hand, [83] does not propose to generate the whole query processing logic in a low level language, like *Low Level Virtual Machine (LLVM)* [72]. Even though generating LLVM assembly provides the opportunity to execute the resulting code using an optimizing JIT compiler provided by LLVM, writing assembler code is more difficult, and we might have

existing database logic implemented in another high-level language like C++ (e.g., index structures). Beside that, in some cases it is not even possible (or desirable) to compile a complex query into a single function, because it might lead to exponential growth in code size [83].

All in all, author of [83] proposes a mixed compilation and execution model, which offers implementing complex parts, that involve complex data structure management or spilling to disk in C++ and that can be pre-compiled. Then, the dynamically generated code in LLVM connects different operators. This scheme results in smaller generated code with very fast compilation time. However, for optimal performance, care should be taken to preserve the hot code paths in LLVM, because while the executable code stays in LLVM, the tuples are kept in the CPU registers. So calling an external function is expensive, as all registers have to be spilled into memory.

2.6 Program Transformation via Staging

Program transformations are categorized into *optimizations* and *lowerings*. An optimization converts given code into more efficient code at the same level of abstraction. A lowering only translates a program into a lower level representation. Optimizations do not have any clear ordering, and hence are prone to phase ordering problems, i.e., the order in which the optimizations are applied impacts the performance of compiled code [68]. So they should be combined for maximum effectiveness. By contrast, lowerings have a clear ordering and undoubtedly should be applied after applying all possible optimizations, in order to avoid missing high-level optimization opportunities [99].

As a result, the direct lowering transformation applied in [83] for generating LLVM or even C++ code and relying on the compiler for optimizations might not produce the most efficient executables, because it might miss some optimizations at the higher levels of abstraction.

To address these shortcomings in program transformation and compilation, extensible compilers were proposed. The goal of these compilers is to reason about user/library-defined programs and data structures. There are two main common mechanisms used in these extensible compilers: 1) Front-end macros, staging or partial evaluation, which all intend to programmatically remove abstractions before they enter the compiler, but might not be able to produce the most optimized code, due to no control over what finally happens in the compiler, and 2) extending internal workings of the compiler by adding new transformation passes at different points in the compilation chain (which leads to phase ordering problems) and new *intermediate representation (IR)* types (whose interaction with the existing generic optimizations if often unclear) [99].

Many computations can be separated into multiple stages based on the frequency of their execution, or the availability of information. Multi-stage programming (MSP) as proposed by Taha and Sheard [113] makes the stages explicit and makes it possible for programmers to delay the evaluation of a program expression to a subsequent stage (thus, staging an expression).

MSP is closely related to partial evaluation [55], which uses the statically known parts of the input data in a program for applying the specializations. There is an important difference

between these two concepts. Partial evaluation strictly specializes programs and usually comes with soundness guarantees. By contrast, adding staging annotations to a program in an MSP language, such as MetaOCaml [12], is simpler for composing staged functions but needs special attention for preventing a change in the meaning of the program and the results of computation [99].

Staging is usually used as a method for program generation. Evaluating a multi-stage program produces an object program that can normally be compiled and executed against real input data. Using this technique is known to simplify the process of generating programs, and with the same approach, it can be used to simplify program transformation. It is straightforward to split any transformation into traversal and generation parts, where staging helps with the generation part. LMS [99] is a single staging extension for Scala. SC [103] is another staging extension for the same language, but consists of multiple stages.

Both LMS and SC go one step further by implementing internal compiler passes as IR interpreters, which are staged themselves and produce a new program as their output. Consequently, they can delegate control back to the program execution for performing the program transformation.

We enumerate the advantages of using staging for program transformation as follows: First, implementing a (staged) IR interpreter is much simpler than creating a IR to IR transformer, which is the common approach for implementing the optimization inside a compiler. Second, optimizations can be added progressively to a program that uses staging. Third, the staged program (or library) can also control the translation and can affect the generated code [99].

2.7 Applying Optimization Techniques

There is no optimization stage after the translation phase in [83]. It is assumed that the result of translation will be further optimized via the target compiler or the execution environment. Trying to perform a translation that is easier to optimize in the pre-execution steps is still considered in their work and had a noticeable impact on the evaluations. For example, *loop inversion optimization* (see section 2.7.5) is used and explained in their code generation without explicitly naming it. However, there is still the possibility of losing some optimizations, due to the lowering that happens in the translation phase [99].

LMS follows a library-only staging approach. In contrast to dedicated MSP languages that are based on quasi quotation, the only means of computational stage distinction in LMS is using types. *Rep[T]* type in the first stage is used for a computation that yields a result of type T in the next stage. Similarly, type T is used in the first stage for computations that yields constant result in the second stage. The normal Scala type system gathers and propagates information about staged expressions and therefore executes a semi-automatic local *Binding Time Analysis (BTA)*. Now, given a program, if type T is already supported by LMS (and only the standard library of Scala is used in the program) no further actions are required and LMS easily generates the code for the next stage.

Then, upon execution of the first stage program including the LMS library, LMS does not directly produce the next stage program in a source code format, but instead, as an intermediate

representation (IR), which is actually a "sea of nodes" dependency graph [99]. Each IR node in this graph is either a constant (*Const*) or a symbol (*Sym*) which contains an operation. The operation in each Sym might contain other Const, Sym, or Blocks (which is a list of statements, each containing a Sym, preserving the ordinal sequence of operations in that block) [99].

Having this IR graph, LMS applies different transformations by traversing the graph. The sequence of "optimization, lowering, optimization" is iteratively applied on the IR graph, until the lowest representation is reached. Finally, code generation is done by means of a traversal on this graph and a function call for the code generation on each node [99].

User programs build their own abstractions and data structure hierarchies as extensions to the language. In these cases, one can easily use the high extensibility and modularity of LMS to extend it and make it work for its own data-structures. IR graph formation and code generation are both modular, one can add the necessary functions for creating IR nodes of their own abstractions, as well as functions for generating the appropriate code for them. Encapsulation of these extensions in a *trait* makes it re-usable for further usages [99]. We name these extensions as *Lifters*, because they make it possible to lift a program structure to be a part of the IR graph.

In addition, LMS also makes it possibly to do manual staging in which users can program functions of type $Rep[A] \Rightarrow Rep[B]$. One important detail here is the difference between types $Rep[A \Rightarrow B]$ and $Rep[A] \Rightarrow Rep[B]$. The former is a staged function object. Macro systems only allow this kind of functions that only allow lifting expression trees. As a result, limits expressiveness, there is no guarantee that higher order functions are evaluated at staging time. In contrast, the latter is a function on staged values, which allows the function parameter to be evaluated and unfolded at staging time, creating greater opportunities for applying the optimization [99].

Compiler optimizations can be categorized into two types: First, *local optimizations* that aim at optimizing a basic block, using only local information in that block. As there is no control flow structure inside a basic block, the analysis costs for these optimizations are low (saving time and reducing storage requirements for applying this type of optimizations) [116]. Second, *global optimizations*, which are also called "intraprocedural methods," try to analyze and optimize whole functions or program blocks. This gives them more information to operate, but it often requires expensive calculations. Moreover, pessimistic assumptions should be made when there are function calls or accesses to global variables, because usually one cannot guarantee that global variables are not altered by other parts of the system that are not considered by the optimizer [116].

Specialization through constant folding (i.e., computing the expressions for which all their parameters are known at compile-time) is already mentioned in this section, and is automatically covered by LMS via having different types in each stage. In the following sub-sections, we introduce other important commonly used compiler optimizations and the way each of them is addressed in the literature.

2.7.1 Common Sub-expression Elimination (CSE)

This optimization intends to prevent redundant computations by performing them only once and re-using the results, whenever it is needed in the posterior computations. LMS applies this optimization by performing an on-the-fly *A-Normal Form (ANF)* conversion.

In this process, every new expression created in the ANF format is assigned to an identification symbol and registered in a map collection. The symbol is returned to be used instead of that expression in the rest of the program. For every successor expression, it is first checked whether it exists among previously registered expressions. Then, if it is found, instead of creating a new symbol, the symbol of the existing expression is returned [99].

2.7.2 Dead Code Elimination (DCE)

In a computer program, there might exist computations that can be removed, either because they do not contribute to the final result, or the control flow does not allow them to be executed. Eliminating the former case saves a lot of unnecessary computations, and the latter produces a more compact executable.

As was mentioned before, every program transformation and code generation in LMS is just a IR graph traversal. Based on this fact, starting traversal from the result symbol, dead nodes are excluded, just by traversing the strongly connected components of this graph [99].

2.7.3 Combining Optimizations (Speculative Rewriting)

Among compiler optimizations, *rewriting*s are preferred, because it is easy to specify them separately, and they do not require programmers to define abstract interpretation lattices. In both LMS and SC, rewrite rules are defined using smart constructors in the Lifter classes.

Many optimizations are mutually beneficial, and in the presence of loops, optimizations need to make optimistic assumptions for obtaining the best results. If we apply optimizations separately, each optimization should effectively make pessimistic assumptions about the outcome of all the other optimizations. Combined analyses avoids the phase ordering problem by solving everything at the same time. In LMS, forward optimizations are applied at IR construction time, while loop information is incomplete. To avoid changing the semantics of an input program, structured loops are used instead of control flow graphs, and dependency information and rewriting is used instead of explicit data-flow lattices [99].

Commonly, rewritings are semantics preserving and usually pessimistic. The idea used in LMS is to drop this assumption. As a consequence, the rewrites are done speculatively and a rollback to a previous state is applied to get optimistic assumptions. The algorithm works as follows: for each confronted loop, all possible optimizations are applied to loop body, without any initial assumptions. Then, by analyzing the result of the transformation, if any new information is discovered that needs new assumptions in previous steps, the transformed loop body is discarded and the original loop is retransformed using updated assumptions. This cycle is repeated until the analysis result reaches a fixed-point, which is the final acceptable transformation [99].

2.7.4 Data Structure Optimizations

High level data structures are the basis of modern programming. At the same time, they can prevent compilers from applying all possible optimizations. For example, in object oriented programming, each new instance of a class is allocated separately. This expense is magnified when allocating an array of objects [99].

As this overhead is non-negligible, a generic struct interface is used in LMS, which is a generic implementation for *struct* types (similar to *struct* in languages like C with common optimizations). In this interface, struct creation is equivalent to creation of a map that relates field identifiers (static) to values (dynamic) alongside a tag field that contains information about data representation. Moreover, field access just looks up the desired value directly from the map, assuming the argument is a struct node. Struct abstraction can be extended to cover sum types (unions) and inheritance, using a tagged union approach by adding a class field to each struct [99].

Array of Struct (AoS) to *Struct of Array (SoA)* conversion [99] is another possible data-structure optimization using a struct interface. It converts arrays of objects into an object containing several arrays, one for each field. This way, boxing/unboxing in JVM is avoided, SIMD execution is made possible, and new chances for eliminating unused parts of a data-structure might occur.

2.7.5 Loop Inversion

This optimization is only a transformation from a *while loop* to an *if block* containing a *do-while loop*. This optimization reduces the number of jumps by two (if loop is executed), improves branch prediction, and enhances instruction pipelining.

It is argued in [83] that branches are cheap if branch prediction works, which means that a branch is taken nearly never or nearly always. *Loop inversion optimization* helps branch prediction by distinguishing between cases that never enter a loop, and cases that executes several iterations of the loop. The author of [83] reports a 20% improvement by using this optimization.

2.7.6 Loop Fusion and Deforestation

Loop fusion is a loop transformation and optimization technique for improving memory locality by combining multiple loops into a single one. Deforestation is another program transformation for eliminating intermediate data structures that are used to compute the final result of a program. Applying fusion after inlining functions usually results in deforestation, as merging operations that traverse elements of a data structure normally eliminate intermediate results.

Several shortcut fusion systems are proposed in the literature to address these optimizations. These systems usually use the *List* data type as their target type to apply their techniques, as it is the primary data-structure for functional programming. List is even used as a control

structure in lazy languages (e.g., Haskell). List operations can be classified as producer (e.g., list constructor), transformer (e.g., map), and consumer (e.g., foldl) operations [28].

Build/foldr [41] is one of the most practically successful shortcut fusion systems. There is a single rule in this system to eliminate adjacent occurrences of the list combinators. Weakness of build/foldr is its failure to handle *foldl* (e.g., sum) which consumes a list using an accumulating parameter, and *zips* that to consume multiple lists in parallel. The intuition behind build/foldr is viewing lists as sequences represented by data-structures, and fusing functions that work directly on the structure of that data.

Destroy/unfoldr [112] is another shortcut fusion system which fails at fusing functions on nested lists (e.g., concatMap), list comprehensions, and filter-like functions which must be defined recursively.

Stream fusion [29] is an additional fusion system that was unsuccessful in correctly fusing *nested lists* and *zips*. Unlike *build/foldr*, both *stream fusion* and *destroy/unfoldr* convert list functions to work on the dual of a list, which is an unfolding or co-structure. However, *stream fusion* uses explicit representation for the list co-structure as the *Stream* type, rather than an implicit one used in *destroy/unfoldr* [28].

A new extension to *stream fusion* [28] is a new approach based on "equational transformation." It has wider coverage than previous shortcut fusion systems, and tries to overcome all their limitations (by covering filter, fold, append, concatMap, list comprehensions, and functions on nested lists).

In fact, the Stream type used in [28] encapsulates an *unfold* by wrapping the initial state and the stepper function inside itself.

$$\textbf{data}\ Stream\ \mathbf{a} = \exists \mathbf{s}.Stream(\mathbf{s} \rightarrow Step\ \mathbf{a}\ \mathbf{s})\ \mathbf{s}$$

$$\begin{aligned}\textbf{data}\ Step\ \mathbf{a}\ \mathbf{s} &= Done\\ &\mid Yield\ \mathbf{a}\ \mathbf{s}\\ &\mid Skip\ \mathbf{s}\end{aligned}$$

Conversion from lists to streams happens using *stream* and *unstream* combinators. Assuming *stream . unstream* as the identity on streams, the only specific optimization used for stream fusion is:

$$\forall \mathbf{s}\ .\ stream(unstream\ \mathbf{s}) \mapsto \mathbf{s}$$

Moreover, having converted the functions over list structures into functions on the stream co-structure, the key trick for creating opportunities to optimize away the intermediate *Step* constructors by composed functions on streams is to avoid implementing the stream producers recursively, and doing only one operation on a stream in each step (via using *Skip* state).

The main idea in the stream fusion system is transforming list functions to expose their structure. Then the actual optimizations (e.g., eliminating the intermediate values) are applied

via general purpose compiler optimizations at compile time. Accordingly, having no control over producing the most optimized program, there is a strong dependency between the stream fusion system and the Haskell compiler[29]. This tight coupling even results in a slowdown for almost half of their evaluation benchmarks[29], because some optimizations are not still supported by the Haskell compiler.

Beside that, there are other deficiencies in the stream fusion system: First, for fusing general recursive definitions, writing stream stepper function is not always easy. At the same time, representation of the control-flow as state (for implementing stream version of the list functions) makes them appear inside-out and hard to understand. Second, fusing more general algebraic data types, like Tree, using the same approach requires a new infrastructure for each new data structure.

LMS introduces a new approach for loop fusion and deforestation. The core loop abstraction in LMS presented in [99] is

$$loop(s)\ \overline{x = \mathcal{G}}\ \{\, i \ \Rightarrow\ \overline{E[x \leftarrow f(i)]}\ \}$$

where s is the loop size and i the loop variable getting values from $[0, s)$. Multiple results $\overline{x}$ can be computed in a loop, each of which is bound to a generator $\mathcal{G}$. Using this formulation, a generator can be a Collect, which creates a flat array-like data-structure, a Reduce($\oplus$), which reduces values with an associative operator $\oplus$, or a Bucket($\mathcal{G}$) operation, which produces a nested collection by grouping generated values by key and applying $\mathcal{G}$ to those with matching keys. The yield statements $x \leftarrow f(i)$ form the loop bodies, which give values to the generators of the current loop or an outer loop. These yield statements are placed in an outer context $E[.]$ that might contain other control-flow structures, like loops or conditionals.

Using this model, it is possible to represent many common collection operations. Since LMS does the subsequent required optimizations itself, after applying fusion, it is guaranteed that all intended optimizations are used for generating the output code. However, not all of the loop fusion ideas presented in [99] (without being mentioned) are completely implemented in LMS. There might be some corner cases and details which would be identified only after the uncut implementation [99].

2.8 Parallel Query Execution

Parallelization can be achieved in several different levels, and there are diverse opportunities in each level.

Block-wise processing, despite its disadvantage of creating additional memory accesses, enables inter-tuple parallelism by processing multiple tuples with a single instruction. This is achieved by using *Single Instruction, Multiple Data (SIMD)* instructions in modern CPUs, which executes the same operation for a block of multiple input data, as long as we can keep the whole block in registers.

Modern CPUs are multi-core (because of the frequency limit for a single core), so there are other opportunities for parallelization in this case. To leverage multi-core processing, nearly

all database systems provide inter-query parallelism, i.e., executing each query on a separate core, which allows evaluating multiple queries at the same time.

In addition, there are new ideas for using multi-core processing to achieve intra-query parallelism, thereby evaluating a single query faster [73]. This is applicable by partitioning the input of query processing operators, processing each partition independently and finally merging the results from all partitions.

The author of [83], without going into the details, claims that they can support intra-query parallelism with nearly no code changes. Their generated code always operates on fragments of data which are processed in tight loops and the result is materialized into the next pipeline breaker. These fragments of data are usually determined by the storage system, and they could as well come from a parallelizing decision. Beside that, only some additional logic is required to merge individual results.

Delite [11] is a parallelization framework for domain-specific languages (DSLs) developed on top of LMS. Its goal is to enable the rapid construction of high performance, highly productive DSLs. In addition to LMS capabilities, it generates an execution graph that targets multiple heterogeneous hardware devices. However, Delite is still in the alpha version, and there is no official release date for it, yet.

3 Compiling Database Applications and Transaction Programs

3.1 Introduction

Can good low-level code for database application programs be automatically generated? Does it regularly require human creativity or is there a methodology that can be automated and systematically applied? Is the day-to-day job of systems programmers fundamentally highly creative or could it be considered routine in the future?

Application development is preferable in high-level languages due to many reasons. One common reason is to have a faster development lifecycle due to the higher level of abstraction that they provide. The first prototype of a data processing application is usually written in a high-level language, which is then used to evaluate the ability of the program to produce the desired results. Then, if the prototype achieves the performance goals required by its users, it can be used in the production environment. Otherwise, the code needs to be further optimized for performance.

The task of optimizing an application program can, in principle, either be done manually by an expert programmer or by an automated tool. Code optimization by an expert programmer mainly takes two forms. The first is to, wherever possible, replace an algorithm or local implementation choice by another one that is more efficient in that particular scenario (specialization). The second is to translate parts of the program to a lower abstraction level (by using a lower-level language or inlining library calls) to create opportunities for further optimizations, resulting in a tighter control of the underlying machine through the program code.

Manual optimization can produce a more efficient system, but it is work-intensive and thus expensive. Code lowering and specialization dramatically increase the maintenance cost of the system due to a substantial increase in the size and complexity of the code. If done by a human, there is no guarantee that all the choices taken by the systems programmer are optimal or even improvements, nor can one rely on a human to apply a given known set of optimization techniques consistently and thoroughly – thus there is no optimality guarantee of any form.

The alternative way of achieving code specialization and optimization is to have it be done *automatically* by a system that satisfies the broad label of compiler or code synthesizer. We

are not referring to the code specialization tasks we have, over the past decades, come to universally trust our mainstream compilers to do reliably, but tasks that currently are performed by systems programmers, even when they use a mainstream compiler. There is currently a good deal of excitement in research circles, mostly revolving around domain-specific compiler technology [32, 64, 83, 103, 110], that suggests that automation of the systems programmer in a domain such as databases that may soon become reality.

In this chapter, we study the compilation of database application programs and propose how to build a compiler-based infrastructure for whole-system specialization. Ideally, using our compiler, database application programmers may use high-level programming abstractions without worrying about performance degradation (*abstraction without regret* as in [65]). The compiler simulates the work of a human systems programmer and employs similar methods and optimizations. The thesis is that the resulting code can match or even outperform the code created by human experts.

For 40 years, since the earliest times of relational DBMS, the compilation of query plans is studied.[i] Recently, we have seen a revival of query compilation in contexts such as stream processing systems (IBM Spade and Streambase) and analytical queries (DBToaster [1], LegoBase [64, 103] and Hyper [83]). However, the optimized compilation of application and transaction code is largely unknown territory, only recently entered by Microsoft's Hekaton system [32], for a limited subset of T-SQL, the domain-specific language (DSL) of Microsoft SQL Server for writing transaction programs. Good speedups compared to classical interpreted evaluation have been reported, but it is still open whether such a compiler can rival the performance hand-optimized code can achieve (cf. [109]) and what is the approximated individual impact of these optimizations.

In contrast to queries, which are phrased in a limited language, database application programs have more variability and mix queries and updates with general-purpose code. Therefore, an application compiler's internal code representations need to be more generic than those of a query compiler. But once such a powerful language is supported, the system can compile, inline and optimize the application programs *together* with generic server code, achieving a maximal degree of specialization and thus performance.

While modern compilers for general-purpose programming languages perform many generic optimizations such as inlining, fusion and deforestation, they do so conservatively. Obtaining maximal performance necessitates moving away from conservative heuristics and pessimistic approaches of the general purpose compilers. In addition, there exist some other optimizations that are in general impossible or meaningless outside the domain of database applications. For example, general purpose compilers treat all function calls equally, even though we could leverage *domain-specific knowledge* (e.g., that joins are commutative). Since these compilers lack domain/data-structure specific optimizations, they are not able to match the performance of code written in a low-level language by a human expert.

[i] In the very beginning of the System R project, query execution was based on compilation. Because the 1970s compiler technology was inconvenient in a rapidly progressing project, this compiler was replaced by the now standard interpretation approach just before the first release. Also many new ideas had to be explored: it was much faster to implement an algorithm directly than to write a second algorithm that outputs the code of the first [17].

Table 3.1 – Optimizations employed in an expertly hand-written implementation of TPC-C [109], and by Beta.

TPC-C transaction	Version	Mutable records	Inlining	Removing index update	Data structure specialization	Index introduction	Hoisting reusable objects	Using runtime info
NewOrder	Beta	✓	✓	✓	✓	✓	✓	✓
	Hand-written	✓	✗	✓	✓	✓	✗	✓
Payment	Beta	✓	✓	✓	✓	✓	✓	✓
	Hand-written	✓	✗	✓	✓	✗	✗	✓
OrderStatus	Beta	✓	✓	–	✓	✓	✓	✓
	Hand-written	✓	✗	–	✓	✗	✗	✓
Delivery	Beta	✓	✓	✓	✓	✓	✓	✓
	Hand-written	✓	✗	✓	✓	✗	✗	✓
StockLevel	Beta	✓	✓	–	✓	✓	✓	✓
	Hand-written	✓	✗	–	✓	✗	✗	✓

Although database application programs can use the control structures of general-purpose programming languages, for the purpose of optimization, the task is greatly simplified – and the automatic generation of expertly optimized code becomes feasible – by the fact that the language used remains domain-specific by its use of a very narrow and known set of data structures. Database application programs tend to use the database's storage structures (tables, or the database connectivity library's proxies – "result sets") as their only dynamic data structures. Creating special compiler optimizations related to such collection data structures can thus be highly effective, particularly if one keeps in mind that domain-specific (abstract) data types are a natural focal point for most domain-specific compiler optimizations.

Beta. We propose a database application compilation suite called Beta[ii]; its novelty lies in its adoption of modern compiler ideas such as *staging* [17, 83]. Optimizations are applied on the input program in multiple stages, and in each stage, parts of the program are partially evaluated based on the arguments known at compile time. Beta also has a runtime engine that is flexible enough to run the generated code in any combination of optimizations. Moreover, the runtime engine exposes the internals of the data structures to Beta, and the latter compiles them together with the application programs.

Inside-Out Data Structures. Data structures designed to be used by programmers usually follow the *information hiding principle*, which means they do not expose their internal implementation to programmers. Even though a compiler does not have to follow the information hiding principle, the analysis for such an optimization (to disregard the additional program-

[ii] Referring to beta-reduction in lambda calculus, which *compiles applications*.

ming layers created only for applying the information hiding principle) is almost impossible for a generic compiler. However, Beta can perform such an optimization because it can reason more easily about the behavior of the limited set of data structures that are used. More details about the design of such an execution engine are described in section 3.5.

In this chapter, we first focus on achieving the best possible performance in a main-memory, single-partition, single-core case where programs are short-running and run in sequence. However, many database application programs run concurrently and in parallel to make the best use of resources available in the system. In such an environment, the concurrency control mechanism used in the execution engine of the database system plays an important role in its performance. It also impacts how a database application compiler, such as Beta, is employed. We later discuss the concurrency aspects in section 3.7. Moreover, to provide durability, Beta makes use of command-logging similar to H-Store [54], but is extensible to accept other mechanisms as well.

Beating the Humans. Beta is inspired by the highly optimized manual implementation of TPC-C that was presented in [109], and which was used there to motivate the creation of the H-Store system (as the implementation is shown to outperform existing OLTP systems by two orders of magnitude). We use this codebase, developed by a group of acknowledged experts, to make headway on the questions we asked at the start of this section. Beta seems to be the natural completion of the H-Store vision: Writing low-level code that achieves the performance called for is very hard and there are few human programmers able to achieve it. Our focus on this use case is also due to the absence of any second such highly optimized low-level codebase in this space that we are aware of.

We acknowledge that the compilation of database application programs is a very broad space, touching many rich fields such as concurrency control and durability, which cannot all be satisfactorily addressed here. Moreover, the challenge can be addressed in fundamentally opposing ways depending on choices such as whether the application code is to be run on the client or the server. We cannot – and will not claim to – offer a design to address all these directions. Instead, we will derive insights from one well-known case study, and see to what extent we can draw generalizable and relevant conclusions from it. The compiler we have created is a work in-progress towards an OLTP system and is meant to generalize these insights (and we evaluate it on a second use case in this chapter), but we do not claim that it provides a solution to all questions related to compiling database application code. In some cases, we make design choices specifically to preserve the case study from [109] as a valid baseline for experimentation.

In our experiments, we show that the compiler, with its superior stamina, even beats the human experts at their own game, making the vision of [109] come true: *a database system with a high-level language performs 100 times faster than a classical RDBMS*, and is competitive with "ideal" hand-optimized code. We demonstrate this for the TPC-C benchmark, evaluating Beta against hand-optimized codebases. Our results show that the TPC-C programs compiled using Beta are twice as efficient as the hand-optimized implementation of [109]. In Table 3.1 we compare, in terms of key optimizations employed, this hand-optimized implementation with the code generated by Beta. Some optimizations do not apply to the TPC-C transaction programs, but our compiler uses all the optimization ideas employed by the human expert and applies them more thoroughly. The details of these optimizations are discussed in section 3.4.

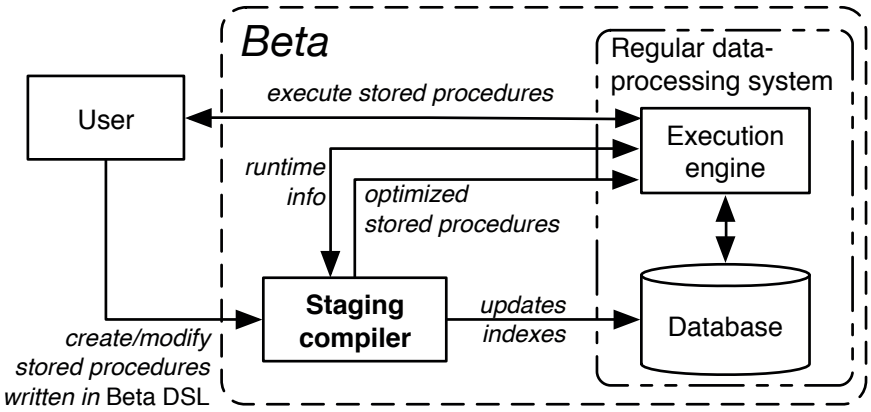


Figure 3.1 – The architecture of Beta.

To get further insights into the generality of our technique, we experiment with incremental view maintenance (IVM) trigger programs. DBToaster [1] generates IVM engines in two steps: SQL queries are transformed into trigger programs (front-end), which are then converted into executable code (back-end). By replacing the back-end with Beta, we gained two orders of magnitude performance improvement over the original DBToaster compiler, which does not perform such extensive optimizations.

In summary, the contributions of this chapter are as follows:

- Beta, a database application execution infrastructure with a staging compiler in its heart. It has an extended DSL that provides a minimal set of features required by database application programmers, on top of which well-defined optimizations are applied. It applies these optimizations more thoroughly and generates code that outperforms the hand-written code by human experts.
- A mechanism for the core supporting data structures to expose their internal functioning to the compiler.
- Identifying key optimizations that are relevant in database applications and detailed experiments quantifying their impact.

We examine the architecture of Beta in detail in the next section, and the structure of the rest of this chapter closely follows the above order of contributions.

3.2 Architecture

The proposed architecture for our domain-specific compilation suite is illustrated in Figure 3.1. We assume that users and applications interact with the database system via stored procedures representing the pieces of application programs: the database application code is moved into the server. This decision not only aligns with the assumption in many novel database systems [109] and minimizes the round-trips between user programs and the database system, but also allows for the necessary code optimizations and the dynamic loading of the freshly

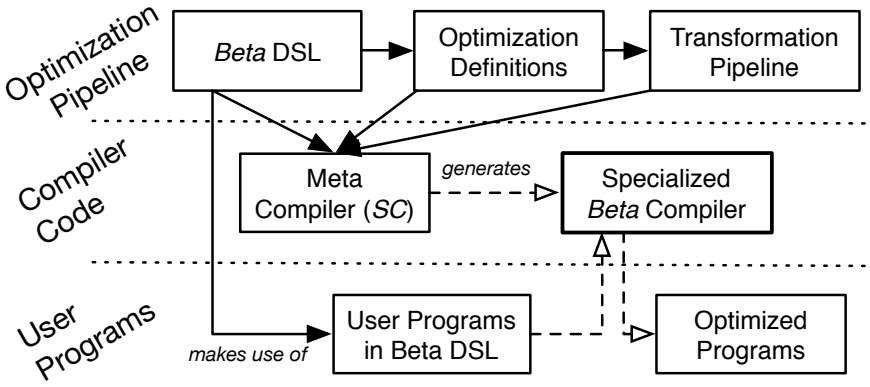


Figure 3.2 – The components of Beta's staging compiler.

compiled code during the operation of the system. In addition, in Beta, with the assumption of having the whole set of stored procedures that are known apriori, the whole application code is compiled together with generic database server code. When these two pieces are put together, the indirections are removed and more optimization opportunities are uncovered.

Beta makes use of an extensible domain-specific language (DSL), called Beta DSL, for writing database application programs. In any domain-specific compiler, the language in which the user programs are written defines the highest level of abstraction given to the compiler. It starts applying its optimizations from that high-level domain. It is therefore essential to have a limited language, but powerful enough to express the application programs. More details of Beta DSL is described in section 3.3.

Beta uses SC [103], a publicly available extensible meta-compiler and staging framework for Scala [85], to compile the programs written in Beta DSL. SC contains an extensible library for applying domain-specific optimizations and code generation for Scala and C++. Figure 3.2 illustrates how SC is used within Beta. Before database application programs can be written, the Beta DSL must first be defined and embedded within the host language, Scala. The next step is to define the domain-specific optimizations and code transformation passes that are applied to this DSL. Given all this information, SC generates the templates for intermediate representation (IR) [34, 100] nodes that will later be used to represent a DSL program during its compilation. The generated IR allows interleaving general purpose constructs (e.g., conditionals and loops) with domain-specific operators (e.g., joins and projections). Beta automatically converts every optimization that is defined on the Beta DSL into functions that manipulate IR nodes representing the program. These optimizations are chained together in the form of a compilation pipeline as shown in Figure 3.3.

To maximize performance, the compiler outputs highly optimized low-level code and all high-level operators are replaced. Accordingly, within the same compiler IR we combine *lowering* and *optimization* passes [100, 103]. Lowering passes convert high-level nodes into their equivalent representation in a more concrete and lower level IR counterpart, while optimizations improve its performance without changing the current level of abstraction of IR graph. Before every lowering, it is necessary to perform all optimization passes in order not to miss any optimization opportunities. For example, join reordering and select-predicate

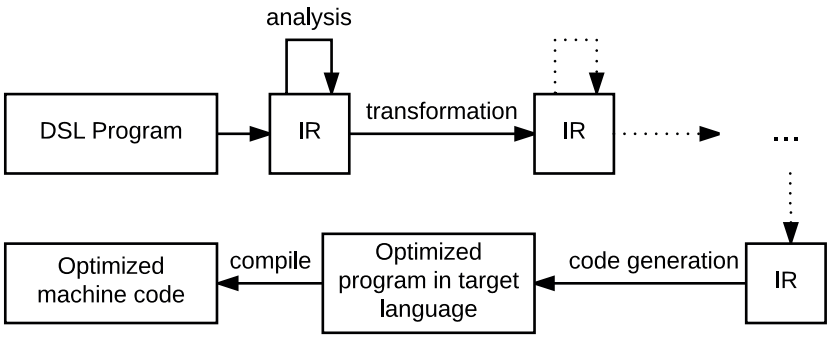


Figure 3.3 – The compilation pipeline for a Beta DSL program.

pushdown are applied before lowering to the appropriate implementation. Optimization passes followed by the lowering passes are applied until the low-level code is generated.

The goal of Beta is to produce highly optimized machine code. To achieve that, complex platform-specific optimizations must be applied for which domain-specific knowledge is useless. The optimizations, such as register allocation, instruction scheduling, cache optimizations are already part of general purpose compilers and Beta leaves these optimizations to a general purpose compiler. Instead, Beta generates code in target languages such as C++ or Scala and gives it to the respective compiler so as to produce the optimized binary. This simplifies the design of Beta while also making it platform agnostic.

Some optimizations that a human expert may suggest are impossible for a compiler to find even in the best of all worlds, simply because they are not *at all times* correct and require some preconditions for their correctness. As an example, the TPC-C code of [109] uses fixed-size arrays to represent metadata tables (such as "items") that do not change after a warmup phase of the benchmark. This optimization contributes significantly to the reported performance improvements over classical OLTP systems. But it remains a cheat unless we find a way for the system to switch between a first metadata loading phase and a second operational phase during which the metadata remains static and only the other tables change. In Beta, as shown in Figure 3.1, we utilize information about the mutability of tables, configuration files, and metadata stored in the tables to produce the fastest possible code. This adds a second, *temporal* dimension to *specialization by design*.

With the meta-compilation, even though the user program can be well modularized, the compiled program is more tightly coupled with the database system code. This is generally not a problem in Beta as the programmers are not going to directly deal with the generated code and therefore, it does not affect the maintainability of the system. However, for any update to a component of the database system or a change in the application programs, we need to compile and deploy a new version. We need to make sure this is not on the critical execution path of the system, so as to avoid complete execution halts between updates.

After the first time that the database application goes into production, any further changes to the codebase go through Beta to create the next specialized version of the application, which is a completely separate process from the running application. Then, the common techniques in

continuous delivery [53] are used to push the updated application into production, assuming the old and new versions of the application access data from a region of memory shared by both. Any change to the underlying system or application program can have two implications: changing the application code and/or underlying data structures and indexes. If it is only the code, then only the new version of the code is dynamically loaded into the execution engine and the new requests are directed to this new version. When the underlying data structures and indexes change, a transition plan is required to move the data into the new version.

The new indexes are created during the application update; after this point, only the instances of new application programs are started; transition then happens gradually until all instances of the old application programs have finished; finally, unused indexes are pruned. If creating a new index is required, a low-overhead snapshotting mechanism is used to capture the current data in the table as well as its subsequent changes during the index creation process, as requests are continuously handled by the application in production. Deleting an index is postponed until all the requests to the previous version of the application are processed. In a distributed setting, an application update could involve re-partitioning the data; this requires replication consistency during the update.

In the following sections, we go into more details of some of these components.

3.3 A DSL for Database Applications

A DSL is a language specialized for a certain domain. DSLs are often high-level and declarative. A declarative interface exposes best the real intent of the programmer by hiding unnecessary details, e.g., loop counters. The key to success for DSLs is to impose a principled way of structuring a program over which the compiler can reason. Recent advances in compiler technology [13, 35, 56] and DSLs [16, 100, 98, 103, 111] are leading to the counter-intuitive insight that automatically generated code can outperform expert handwritten code in some domains [90, 125].

In Beta, we focus on whole application programs; queries are just a part of them. Moreover, it uses a single language that unifies data manipulation and query operations. Having a unified language allows: *i)* consistent syntax, *ii)* unified compiler optimizations for queries and data manipulation programs. Our DSL (Figure 3.4) is embedded in the Scala language [85], which offers an infrastructure for creating and optimizing DSLs [103]. The Beta DSL has one dynamic data structure for tables (i.e., `Store`) and contains two parts:

- High-level functions that operate on relations and provide the same expressive power as SQL (with the Scala syntax).
- A low-level imperative DSL used for implementing the high-level query operators.

The high-level operations can be expressed with the low-level DSL. Yet, distinguishing the two layers is important as some optimizations can only be applied at a higher level, but the programmer can choose to combine them. We now describe the DSL operations and how to *lower* the high level operations.

```
trait Store[T] {
  // high-level operations
  def filter(f:T=>Bool): Store[T]
  def map[U](f:T=>U): Store[U]
  def fold[U](zero:U)(f:(U,T)=>U): U
  def join[U](s:Store[U],c:(T,U)=>Bool):Store[(T,U)]
  def groupBy[K](p:T=>K): Store[(K, Store[T])]
  def union[T](s:Store[T]): Store[T]
  // low-level operations
  def get[K](p:T=>K, key:K): T
  def slice[K](p:T=>K, key:K): Store[T]
  def range[K](p:T=>K, from:K, to:K,
               options:RangeOptions=DEFAULT): Store[T]
  def foreach(f:T=>Unit): Unit
  def insert(t:T): Unit
  def update[K](p:T=>K, key:K, updated:T): Unit
  def delete(t:T): Unit
}
```

Figure 3.4 – Interface for the Beta DSL.

High-level DSL. A high-level DSL gives more freedom to the compiler to choose the best implementation. SQL is one such a DSL for queries where the join implementation is left to the query optimizer.

Our high-level DSL is similar to Monad Calculus [10] and collections in LINQ [77]. Operation `groupBy` assigns tuples to groups (`p:T=>K`). Operation `join` joins two relations and accepts a boolean function as join condition that, given a tuple from both relations, determines whether this pair is in the join result. The `map` method is used for transformation (e.g., projection), `filter` for selection, `fold` for aggregation, and `union` for combining two relations.

A query expressed in the high-level DSL is a sequence of function calls over `Store` objects. This representation is convenient to rearrange operators (it is much harder in the low-level DSL). For example, we can apply classical query optimizations on this high-level DSL. These optimizations are typically done by a query optimizer and have been studied for more than forty years [107].

Low-level DSL. The low-level DSL contains fine-grained operations on the `Store` objects. Selecting a single record in a unique index is done via `get`, which accepts a function (`p:T=>K`) to extract the search key from each record, as well as the target search key (`key:K`). It should be guaranteed by design that the given key to `get` is a unique key. Operation `slice` is similar to `get` but used for cases where the given search key is not a unique key. Operation `range` applies range queries and the `options` argument controls the inversion of the range and bounds inclusion. Operation `foreach` applies a function to each record of a `Store`. Finally, `update`, `delete`, and `insert` modify the relation content.

The low-level DSL expresses query plans. Its interface forbids less efficient functional style on `Stores`[iii]. The only supported operations are loops or operations on individual elements. Although high-level data manipulation is functional (immutable data), the low-level DSL uses mutability for in-place updates with lazy re-indexing of the tuple in the `Store`. The low-level DSL still abstracts away indexes: the projections `p:T=>K` define the potential partition columns, but do not impose an implementation. Indexes can encode complex operations that are further optimized for each phase specifically (see section 3.4.2). For example, in the TPC-C `Delivery` transaction program,

```
SELECT o_id FROM new_order WHERE d_id=:d
  AND w_id=:w ORDER BY o_id ASC LIMIT 1;
```

corresponds to the `slice` operation followed by a minimum aggregator [iv]:

```
newOrder.slice(o=>(o.d_id,o.w_id),(d,w))
        .min(o=>o.o_id)
```

and the corresponding index could be implemented with a hash-table of heaps.

Transition from high-level to low-level DSL. Once optimized, the high-level DSL is converted into low-level operators by the following rules:

- *Selection (filter)*: There are two cases for *filter*: if only one element is expected, we retrieve it with a `get` operation, otherwise we iterate over all the elements matching the predicate using `slice` for equality predicates, and `range` to address all equalities and up to one inequality. If additional predicates exist, they can be applied to the output of `slice` or `range` operations. Exposing these specific iteration operations helps generating specialized indexes (cf. section 3.5).
- *Fold, projection (map), groupBy, union*: accumulate into a mutable variable or a regular Scala collection which then acts as a source.
- *Join*: is converted into nested loops depending on the join predicate (`foreach`, `range`, `slice`, and possibly `get`).

Although the transformations are straightforward, combining them efficiently is not trivial. As an example, consider the following SQL query (in the absence of NULL values) and its counterpart program:

```
SELECT SUM(p) FROM t WHERE c1=v1 AND c2<v2
```

```
t.filter( x => x.c1==v1 && x.c2<v2 )
     .map( x => x.p ).fold(0, (x,y) => x+y )
```

This program is lowered optimally using a range index; alternatively, we could use a hash index and test the rest of the filter predicate in the closure:

[iii] Automatically transformed by deforestation, cf. section 3.4.

[iv] The `min` operation is implemented as an implicit conversion to an extended Beta DSL that supports it, even though `min` and other similar operations could be a part of the Beta DSL and be implemented using the `foreach` operation.

```
// using range (tree-like)
var sum=0
t.range( x => (x.c1,x.c2), (v1,-INF),(v1,v2) )
       { x => sum+=x.p }
// using slice (hash-index)
var sum=0
t.slice(x => x.c1, v1,{ x => if (x.c2<v2) sum+=x.p })
```

Although high and low-level DSLs respectively bear resemblance with queries and query plans in the way they separate concerns, there exists a key difference that we can combine them and reuse the same constructs (control flow) across DSLs; the compiler can optimize across domains. In the next section, we detail the optimizations that take place during the DSL conversion.

3.4 Optimizations

We describe the key optimizations to achieve optimal transformation from a functional DSL into imperative programming (low-level DSL). This set of optimizations was obtained by reverse-engineering the TPC-C implementation of [109] and listed in Table 3.1. We do not discuss common optimizations like dead code elimination (DCE) [80], common subexpression elimination (CSE) [80] on both domain-specific operators and low-level constructs, e.g., removal of duplicate projection, selection, and joins, and loop-invariant code motion (computations are moved outside of loops when possible) [80]. These optimizations are well studied and SC [103] provides them out of the box. Also, we do not cover high-level language optimizations; these are covered by classical query optimizers and are well studied elsewhere.

3.4.1 Removing Intermediate Materializations

Composition allows us to write complex programs easily but it comes with a cost.

Example 3.1. In a simple selection-projection query:

```
customers.map(x => (x.credit-x.bal,
                    x.name.substring(0, 5))
        ⊙.filter((credit, name) => credit < 47000)
        ⊙.map((credit, name) => name)
        ⊙.fold(0)((x, acc) => acc + 1)
```

each function iterates over the whole collection, and at each junction point (⊙) an intermediate collection is materialized (in memory). △

In the above example, we map the original relation into a new collection. Then, we create a new subset of these elements, when we are only interested in the number of customers within a credit cap. Creating intermediate collections incurs large performance penalties, especially when the amount of computation per element is small.

Deforestation with staged data structures. To remove materializations without changing the programming model, Beta applies *deforestation* (or *fusion*) [126, 42] by using the techniques described in [42].

Example 3.1 (continued)**.** If `customer` is implemented as an array, the final program after `deforestation` would become:

```
var res = 0 //accumulator of the fold
val len = customers.length
var i = 0
while (i < len) {
  val x = customers(i)
  // first map result
  val y = Entry(x.credit-x.bal, x.name.substring(0, 5))
  if (y._1 < 47000) { //filter
    val z = Entry(y._1) // second map result
    res += 1 //fold
  }
  i += 1
}
return res
```

△

Similar optimizations are applied to table joins, index operations, etc. For example, Beta replaces relation joins by their appropriate join implementations.

Removing temporary records. Deforestation provides great performance improvements but cannot completely avoid materialization between function calls: an intermediate structure is constructed as the return value and immediately destructed by the next call.

Example 3.1 (continued)**.** The value y is created although the values of x could be used directly. In the body of the loop we expect to obtain:

```
while (i < len) {
  val x = customers(i)
  if (x.credit-x.bal < 47000) res += 1
  i += 1
}
```

△

In order to remove intermediate materializations, one needs to be aware of the characteristics of the intermediate results (tuples). As creating intermediate results may have side-effects (e.g., a function passed to tt map that also modifies a global variable in its body), this optimization requires data flow analysis and is not generally applicable. Intermediate materializations can be eliminated in two ways:

- By using continuation passing style (CPS) [3] where only relevant intermediate results are evaluated and used.

- By breaking up structures (tuples) into individual variables, eliminating unused code, and fusing variables back together in merge-points of the control flow graph. This approach is taken by Beta.

3.4.2 Automatic Indexing

In traditional databases, any SQL query could potentially be executed by a client. Therefore all tables could be mutated and automatic indexing is not directly possible without further assistance from the users, probably via user-specified annotations. Choosing appropriate indexes is a difficult task that is left to a human operator, usually a database administrator, possibly guided by query optimizer hints and heuristics-based tools [20].

Beta gathers all the static information available in the database application and combines it with runtime information from the execution engine to guide the automatic indexing mechanisms, and automatically introduce indexes for the tables. With Beta, this is possible since, at any point in time, the set of possible programs is closed as they are defined by the current application. Furthermore, in Beta DSL, all table accesses and writes are explicit and it is trivial to collect them. Given this closed world of programs and information about access patterns, Beta infers the optimal indexes.

In the rest of this section, we talk about one such automatic indexing mechanism that is used by Beta, but essentially the information gathered by Beta can be fed into any other automatic index advisor tool [20]. Then, after proposing the proper indexes for an application, Beta takes care of the necessary program transformations to employ these indexes, such as generating the proper hashing and comparison functions required by indexes.

Analyzing access patterns and data mutation. The range of possible data structures for indexes is limited by the data mutations: if records are inserted or deleted, an immutable index (e.g., fixed-size array) can not be used; in the presence of updates, it depends on whether modified columns are part of the index. To create efficient indexes, we are interested in the access patterns: partitioning (`slice`), range scanning, minimum/maximum selection and uniqueness (primary or unique key).

In Beta DSL, access patterns are detected based on the chain of Store operations (Figure 3.4). For example, `get` retrieves an element in a unique index and `range` denotes a range selection. The key insight is that indexes can be automatically generated during the compilation process. Beta first searches for access patterns and mutations in the intermediate representation of all programs. Each relevant operation is registered with the associated table. After the analysis, every table in the phase is marked with mutability flags (insert/delete/update), and a list of candidate indexes (pattern and table columns). This information is then used to select indexes; heuristics are studied in the database literature [48, 19].

Access index selection. For in-memory databases, the clustering and sequentiality (reminiscent of disk-based DBMS) are less relevant as all locations can be accessed in constant time (without a seek penalty). Therefore, we can simplify primary and secondary indexes design by retaining only their uniqueness property.

Table 3.2 – Decision-table for automatic index creation after program analysis in Beta.

Store Operation	Equivalent Index
<dynamic table> get	Unique hash
slice	Non-unique hash
range	Hash of B+ trees
min or max	Hash of binary heaps
median	Hash of two binary heaps (min/max)
<static table> get	Hash lookup Array
(none)	Linked list

To store and lookup data, Beta relies on the most appropriate data structure: arrays, lists, heaps, B+trees, hash-sets, etc. Each structure has different access time guarantees for lookup, iteration over a subset or all elements, and maintenance costs (in presence of update, insert and delete).

The index creation strategy of Beta is shown in Table 3.2; our insight is that we can nest well-known data structures into each other. For example, we obtain the minimum of a partition of the data using a heap nested in a hash-table. In a read-prevalent workload, this strategy always pays off because indexes are more often read than updated. A detailed example of applying the `Automatic Indexing` on TPC-C benchmark is provided in section 3.4.3.

3.4.3 Automatic Indexing Example

We use the TPC-C transaction programs to illustrate the automatic index creation. The TPC-C benchmark consists of five transactions and the workload is a mix of them with specific ratios. Two of them (OrderStatus and StockLevel) are read-only, others (NewOrder, Payment, and Delivery) also write into the database. The table schema of TPC-C is shown in Figure 3.5.

The static analysis of TPC-C transactions shows that Warehouse, District, Item, Customer, and Stock tables do not grow or shrink in size. Beta uses this information to convert the indexing mechanism for these tables to an array index over one (Warehouse, Item) or multiple columns (District, Customer, and Stock tables).

This conversion improves the performance: the array position is obtained with a numerical operation. The ranges of indexed columns are also required; these are obtained by inspecting the data. Beta maps the indexed columns into array position as follows:

- If the range (product) is much larger than the number of entries, we use a hash-table
- If a single column is indexed, its values must be unique
- If the index uses multiple columns, their combination must be unique, and therefore, Beta constructs a mapping from the cross-product of the ranges to an integer value.

For the rest of the tables, most of the queries are either lookups on the primary key or contain equality conditions. Hashing is, therefore, the preferred indexing method. The index columns are determined by the key of `get` and `slice` operations. For queries containing inequality

conditions, Beta chooses between iterating the whole map when selectivity is low and using a B+ tree index when it is high. Additionally, a B+ tree index (possibly nested inside a hash-index) addresses all equalities and one inequality condition, and other conditions are verified on the returned tuples. Similar rules apply for minimum, maximum and median queries. Beta uses a hash-table of min or max-heap indexes for faster lookup of minimum or maximum element in a partition, respectively. Moreover, a hash-table of two combined balanced heaps (one min-heap and one max-heap, where their sizes differ by at most one) is used for efficient lookup and maintenance of the median element in a partition.

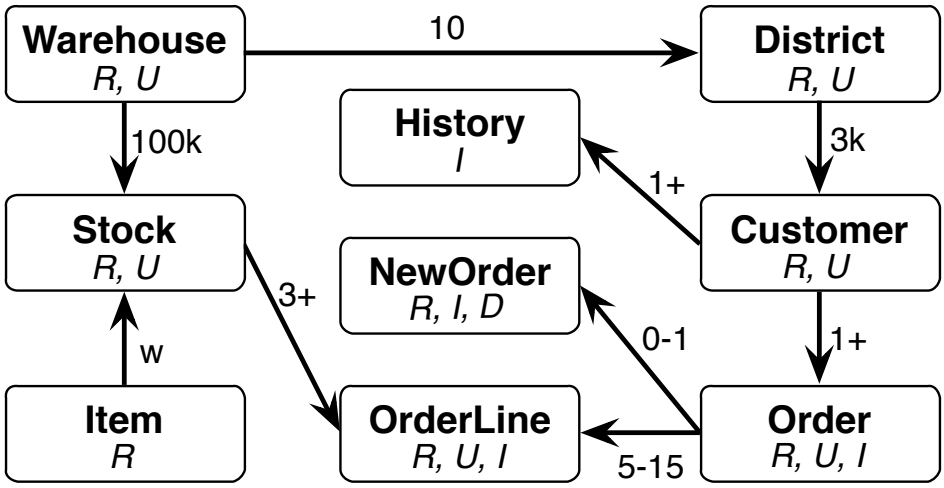


Figure 3.5 – Entity-relationship diagram for TPC-C benchmark and operations of the transactions on each table: R:read, U:update, I:insert and D:delete.

Example 3.2. Consider a simplified excerpt of the Delivery transaction for TPC-C benchmark in Beta DSL:

```
for (d_id <- 1 to DIST_PER_WAREHOUSE) {
  val top_order:NewOrderEntry =
     new_order.slice(x=>(x.d_id,x.w_id), (d_id,w_id))
              .min(x=>x.no_o_id)
  orderIDs(d_id - 1) = top_order.o_id
  new_order.delete(top_order)
}
```

The generated code contains two parts: the specialized data structure for `NewOrder` tuples (NewOrderEntry) and the executable program code. The `NewOrder` table is provided with index-specific methods for comparison and hash code. The generated code with reusable keys and automatic indexing (*without inlining*) is:

```
class NewOrderEntry(var o_id:Int=0, var d_id:Int=0,
                    var w_id:Int=0) extends Entry
type E = NewOrderEntry
object NO_Idx extends IndexFunc[E] { // slicing
  def hash(e:E) = do_hash(e.d_id,e.w_id)
  def cmp(e1:E,e2:E) = if (e1.d_id==e2.d_id &&
```

```
                                e1.w_id==e2.w_id) 0 else 1
}
object NO_Ord extends IndexFunc[E] { // ordering
  def hash(e:E) = 0 // unused
  def cmp(e1:E,e2:E) = signum(e1.o_id-e2.o_id)
}
val new_order = new Store[E]()
// index(1)=SliceMin: slice=NO_Idx, cmp=NO_Ord
new_order.index(NO_Idx, SliceHeapMinIdx, NO_Ord)
val param = new NewOrderEntry //parameter holder

// executable program: gathers the oldest NO for
// each district and removes it from new_order table
var x1: Int = 1
do {
  param.d_id=x1; param.w_id=$w_id;
  //get the minimum NO using the first index
  val x2 =  new_order.get(1, param)
  val x3 = x2._1
  val x4 = x1 - 1
  orderIDs(x4) = x3
  new_order.delete(x2)
  x1 += 1
} while (x1 <= 10)
```

A hash-index is created over (w_id, d_id); each bucket is a binary-heap where the order is given by the o_id column. Other optimizations in the above code are: reusable keys (`param`) and record structure specialization (`NewOrderEntry` class is generated). △

In practice, our greedy strategy applied to 22 TPC-H queries created only 255 indexes for 176 relations involved in these queries (a single index for 107 relations, two indexes for 59 relations, and three indexes for 10 relations). Collecting information about relative frequencies of index operations could help in other scenarios.

3.4.4 Hoisting

In many programs, there are temporary objects that either have a complex internal implementation that is not worth inlining or require using a considerable amount of initialization or dynamically allocated memory. In these scenarios, Beta does not see any benefit in inlining and co-optimizing these helper objects with the user-code. However, it re-uses these objects across multiple program executions by moving the helper object into the global scope if possible. Examples of such helper objects that can be hoisted include the temporary objects used for storing the search key fields to be passed to `get` or `slice` operations.

3.4.5 Partial Evaluation

Partial evaluation [55] is a technique to achieve code specialization by using partially known program inputs to generate a new version of the program specialized for those inputs. Using the statically known input (e.g., configuration files), Beta partially evaluates an input program and converts it to a specialized program.

For example, in regular expression matching, the pattern is usually a constant string. The regular expression matcher can be partially evaluated and, when combined with `Hoisting`, moved out to the global scope. String formatter is another commonly used example where the format is normally a constant string. The string formatter can be partially evaluated and replaced with specific low-level string operations for that particular format. Then, it is called efficiently many times during the application lifetime, without the necessity to parse and execute the pattern during the execution. Consider the following line of code from the `Payment` transaction of the TPC-C benchmark.

```
1 val h_data="%.10s    %.10s".format(w_name, d_name)
```

After `Partial Evaluation`, the low level C code generated by Beta is the following:

```
1 char h_data[24];
2 strncpy(h_data, w_name, 10);
3 strcat(h_data,"    ");
4 strncat(h_data, d_name, 10);
```

3.4.6 Record Structure Specialization

In Beta, by default, data is stored in the form of general purpose records implemented as a hash table that maps column numbers to untyped data values. The records have to be general purpose, as the database contains several tables, each of which may contain a different number of columns of different types. However, after enabling `Record Structure Specialization`, Beta generates specialized data structures for records of each table in the database. These specialized records contain the exact number of columns of the appropriate types for the table. Each column can be accessed directly, and as the correct type, without requiring a hash table. Moreover, Beta rearranges the memory layout of the records to group the key fields together. This optimization results in better data locality and more efficient memory access as one level of indirection is removed.

Beta first infers the schema of the table from the data loading operations. The schema comprises the list of types of individual columns of the corresponding table. Once the schemas of the tables are known, Beta invokes a type inference logic to associate each instance of records in the application program with the correct schema. It is based on the table whose `Store` instance produced/used the record instance in one of its methods. Finally, the generic record instances are replaced by the specialized ones, along with all associated methods (i.e., getters and setters). In addition, Beta automatically generates the code for defining these specialized data structures, along with equivalent hashing, equality and comparison methods.

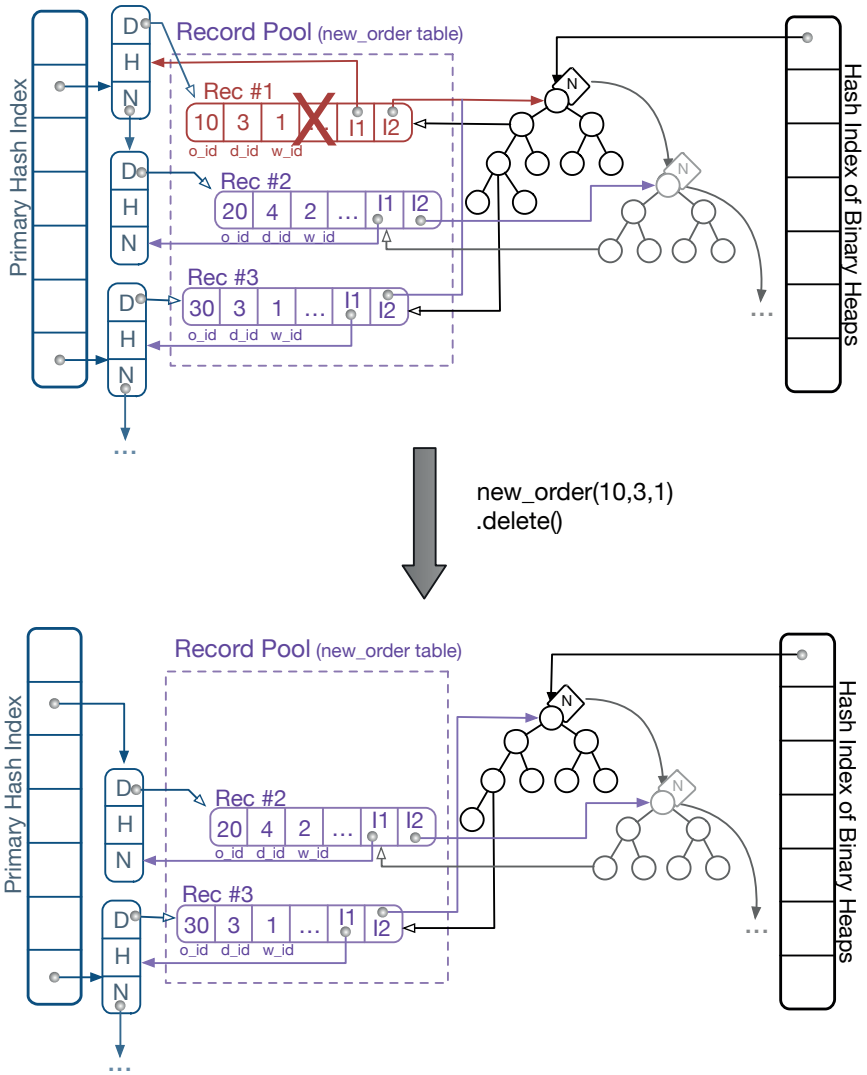


Figure 3.6 – A sample schematic of multi-indexed data structures (IODS) generated by Beta.

3.4.7 Mutable Records

Typical database engines usually return a copy of the values when queried by a user for a multitude of reasons [101]. Any changes to these values are not reflected back to the database unless followed by a subsequent update operation. Consider the following code snippet from the `Delivery` transaction in TPC-C:

```
1 val order=orderTbl.get(t=>(t.no_o_id,t.d_id,t.w_id),
2                          (no_o_id, d_id, w_id))
3 order.o_carrier_id = o_carrier_id
4 orderTbl.update(t => (t.no_o_id, t.d_id, t.w_id),
5                      (no_o_id, d_id, w_id), order)
```

In the above code, an `Order` is looked up, gets updated and is stored back in the table. There are several optimization opportunities: 1) the first line returns a copy of the `Order` record, 2) the third line looks up the `Order` record again to update it, and 3) the update does not affect any index as the updated field is not indexed. Compiled by Beta, the equivalent compiled code is the following:

```
1 val orderRef = orderTbl.getRef(t => (t.no_o_id,
2                t.d_id, t.w_id), (no_o_id, d_id, w_id))
3 orderRef.o_carrier_id = o_carrier_id
```

In this optimized code snippet, all the previously mentioned optimization opportunities are used. The first two opportunities are discussed here, and the last one in the next section.

In short, Beta avoids copying values and subsequent lookups by returning a reference to the actual indexed data instead of a copy when it is safe to do so. In this example, after analyzing the input code, Beta finds out that the `Order` record is used only for updating a non-key field, so it uses `getRef` instead of `get` to directly get a reference to the internal `Order` record. Subsequently, the update on `o_carrier_id` gets applied directly to the `Order` record. Generally, update operations still have to be invoked on the table to update any index meta-information; but this can be done now without another lookup owing to back-references (see section 3.5) employed by Beta. However, in this example, the `update` operation is also completely eliminated, by another optimization explained next (section 3.4.8).

3.4.8 Removing Dead Index Updates

Each `update` operation on a Store results in the propagation of the update to the underlying indexes created for it. However, doing all these updates is not always necessary as not all indexed columns are altered every time. For every update operation, at compile time, Beta tracks which columns are modified. As it knows what columns are indexed by each indexes, it replaces the update operation on the table with those on only the relevant indexes. Moreover, with the `Mutable Records` optimization (section 3.4.7), the primary index is not updated as well if the primary key columns are not updated, and the entire update operation on the table is avoided altogether as in the example above.

3.5 Inside-Out Data Structures

The execution engine of Beta requires a data structure that implements all the operations in the Beta DSL (section 3.3) and supports all the optimizations described in section 3.4. Beta uses a multi-indexed data structure, called IODS, that exposes its internals to the Beta optimizations and does not follow the information hiding principle as in existing alternatives. The exposed internals help Beta to use IODS effectively. An IODS is an implementation of the `Store` in the Beta DSL and is composed of a record pool and one or more indexes defined on top of it. All the indexes defined on an IODS share the same records from the pool. Beta knows how these records are used by the application programs as well as by IODS. In addition, these records are aware of their memory layout within IODS at runtime.

The mechanism of sharing records among indexes taken in IODS is different from the mainstream approach of separating keys and values. The benefit of this approach is that unnecessary data copies are avoided and there is a single location to apply all the updates to the data, even though the index meta-information still needs to be updated. It may appear that, by not separating the key for indexes, Beta loses data locality for the hash and comparison functions on the key. However, this is not the case, as `Record Structure Specialization` (section 3.4.6) takes care of putting the key fields near each other in the memory layout for the record structure. Another disadvantage of the record pool approach is that indexes can no longer use generic hashing and compare functions, which are more convenient for the human programmer as the key fields vary between different types of records and indexes. On the other hand, if specialized hashing and compare functions are generated automatically, then the performance is improved by tuning these functions for their respective usages. This optimization is done nonetheless by Beta and, therefore, this is not an additional overhead for the record pool approach.

Beta takes advantage of the knowledge of how the underlying data structures are used in two ways. Firstly, it applies specialized data structure tuning by removing the unnecessary features of the data structure, or replace it with a better one. IODS relies on Beta to pick the most appropriate indexes for the application, which is explained in `Automatic Indexing` (section 3.4.2). Secondly, it takes over some parts of the implementation and bypasses the data structure as in the case of optimizations such as `Mutable Records` optimization (section 3.4.7).

By having records that contain meta-information regarding their position within the indexes, subsequent index operations on the same record are performed without having to look them up again. To realize this, the records in Beta have back-references to their corresponding *index containers*, i.e., the closest sub-structure in the index that has a reference to the record. Moreover, each read operation from an index returns references to one or more of these complete records, instead of a copy, when instructed by Beta. This works in conjunction with the knowledge of how a particular record is used within the application program.

As Beta knows how each index handles individual records, it efficiently applies the `Mutable Records` (section 3.4.7) and `Removing dead index -updates` (section 3.4.8) optimizations. For an update operation, Beta keeps track of modified columns at compilation time, and updates only the indexes referring to these columns. In addition, the `update` and `delete` operations become much cheaper to apply, as the indexes avoid any further lookup by using the back-references. For example, a delete operation in an index backed by a (doubly) linked-list becomes an O(1) operation instead of O(n) that would be required to look it up again.

Figure 3.6 illustrates how table records are referenced by multiple indexes. This IODS is generated for the `NewOrder` table in TPC-C benchmark, and is automatically created based on the access methods on this table throughout the benchmark. Each cell is labeled with a letter, D, H, N, or I, which correspondingly stand for Data, Hash, Next pointer, and Index. The three fields in each `NewOrder` are *order ID* (*o_id*), *district ID* (*d_id*) and *warehouse ID* (*w_id*). The primary index is created on all of these fields and is chosen to be a hash-index. The secondary index is created on *d_id* and *w_id* and is a hash-index of binary heaps, where each binary heap maintains the minimum `NewOrder` record ordered by *o_id*. Each `NewOrder` record stores back-references to its corresponding index container.

Figure 3.6 also shows how a `delete` operation on a `NewOrder` record benefits from the available back references to the index containers for applying the operation without a prior lookup into the indexes. In this example, a `NewOrder` record is first looked up from its secondary index, and with the `Mutable Records` optimization enabled, a reference to the actual record is returned. It is this reference that is passed to the `delete` operation on the IODS, which in turn passes it to the `delete` operation on the indexes. Since each index receives the actual reference to the entry to be deleted, it can use the back-references to find its container for that record. For the primary hash-index, the container forms a part of the doubly linked collision chain for that hash bucket, and removing the container is straightforward as in any doubly linked list. For the secondary index, the container is part of a heap that resides in a hash bucket. Its deletion requires a lookup in neither the hash-index nor the heap, and is a standard remove operation on the heap.

3.6 Evaluation

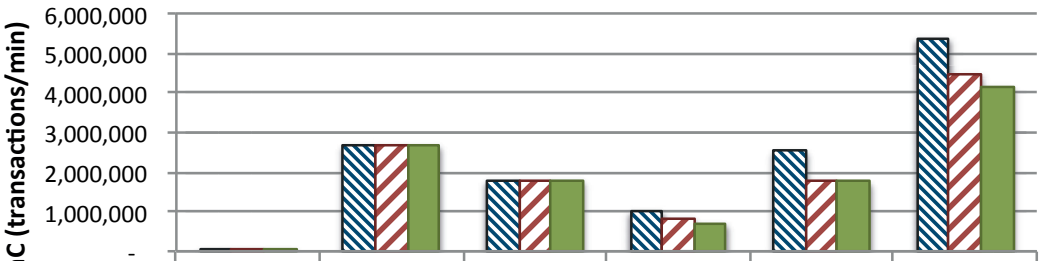
To evaluate Beta, we use two benchmarks: TPC-C and trigger programs generated by DBToaster [1] (r2827) for the incremental view maintenance of TPC-H queries. All experiments are performed on a single-socket Intel® Xeon® E5-2620 (6 physical cores), 128GB of RAM, Ubuntu 16.04.3 LTS, GCC 4.8.4, and Oracle Java(TM) SE Runtime Environment (build 1.8.0_111-b14). Hyper-threading, turbo-boost, and frequency scaling are disabled to achieve more stable results.

We used SC 0.1.4, running with Scala 2.11.2 on the Java Hotspot 64-bit server VM having 64 GB of heap memory, and the generated code was compiled with the `-optimise` option of the Scala compiler and `-O3` for GCC. We run all benchmarks 5 times and report the median of last 3 measurements. Variance in all experiments is less than 3% so we omit it from the graphs.

3.6.1 Performance Model

To evaluate Beta, we study the impact of individual optimizations by enabling different combinations of optimizations on the same application. However, to compare the impact of individual optimizations, we need a base case. Naturally, the first thing that comes to mind is to compare the scenario with only one optimization against the unoptimized application, for each optimization. We consider the ratio of throughput of the application after applying the optimization to that of the original application, as the impact of that particular optimization. To be precise, if t_0 is the total time for executing the unoptimized application, t_X is that after applying optimization X, and s_i is the speedup provided by X through optimizing a sub-program (block) b_i that constitutes a fraction f_i of the total time t_0 of the unoptimized application, then $t_X := t_0 * \sum \frac{f_i}{s_i}$.

The impact of optimization X is given by $\frac{t_0}{t_X}$, as the ratio of throughputs is the inverse of that of execution times. We infer that 1) keeping the fraction of blocks f_i constant, if the speedup s_i is increased, the overall impact of the optimization also increases, and 2) increasing f_i for blocks with large speedup s_i also increases the overall impact due to the restriction that $\sum f_i$ is one. In other words, the higher the speedup s_i and the fraction f_i of blocks with high speedup, the higher would be the impact of the optimization.



	Scala Naïve	C++ [109]	C++ [109] Revised	Scala HW	Beta Scala	Beta C++
W=1	747	2,663,092	1,795,411	1,017,520	2,535,159	5,362,123
W=5	178	2,673,561	1,795,910	846,982	1,787,235	4,444,587
W=10	78	2,670,703	1,795,614	709,949	1,756,347	4,126,893

Figure 3.7 – TPC-C benchmark results. Comparison is made based on TpmC for different scale-factors (W). TpmC is the primary metric for TPC-C benchmark, which denotes the number of new-order transactions per minute.

However, this model does not do justice to some optimizations that are crucial for the most optimized code but are insignificant in the original application. This happens because the impact of an optimization depends not just on what it can do (s_i), but also on where it can be applied (f_i). For example, a block of code b_k can have a small fraction f_k in the unoptimized code, but as other optimizations are applied, f_k can increase significantly if these optimizations improve other blocks and leave b_k relatively untouched. At this point, the impact of applying some optimization Y that improves b_k would be much higher compared to when it was applied in the original application. This phenomenon is orthogonal to that where two or more optimizations have an enabling or a disabling effect on each other because a particular block can be optimized by all of them. In view of this, we chose the application obtained after applying all the optimizations to be the base case. To study the impact of an optimization X, we disable it and compare the throughput with that of the most optimized case.

3.6.2 TPC-C Benchmark

We implemented TPC-C transaction programs by a syntactic rewrite of SQL into Beta DSL. The conversion is not yet automated but the correspondence is straightforward (see section 3.3). The TPC-C benchmark specifies the meta-data characteristics and the input parameters for each transaction. This includes the relative size of each table and the expected number of records returned by a query. An expert programmer can, therefore, overuse this information to build a fixed-size implementation that is not otherwise correct. Conversely, we use Beta to infer this information automatically and adapt the compiled program to the underlying data via adding annotations to the program. The results presented in this section are for the single-threaded implementation; those for the multi-core experiments are presented in section 3.7. We compare the code generated by the Beta compiler with the expertly-written

implementation of [109] and other implementations mentioned below. All benchmarks are restricted to single-site execution. The benchmark is run with eight million transactions.

- *Scala naïve*: a direct conversion from SQL to Scala using only the Scala collections without any indexes on the tables.
- *C++ [109]*: the C++ hand-optimized code of [109]. As mentioned in [109], slight changes to the TPC-C benchmark were necessary; we applied the same change-set to other queries to have a fair comparison.
- *C++ [109] Revised*: we corrected all the bugs of [109]: converted static structures to allow experiments longer than one second, corrected the faulty implementation of B+Tree, enabled a costly and necessary condition branch (Payment), replaced a dummy value by an actual minimum lookup (Delivery), and added a count-distinct (Stock-level).
- *Scala Hand-written*: the counterpart implementation of C++ Revised in Scala with hand-optimized code and manually tuned indexes.
- *Beta Scala/C++*: the program resulting from staged compilation. The Beta input transaction code is the same as *Scala naïve*, except it uses Beta DSL instead of standard Scala collections. The compiled programs for both Scala and C++ output languages are measured.

We also compared it against other OLTP systems and found that they were one to two orders of magnitude slower. However, we do not present the results here as it is unfair to compare the code generated by Beta to full OLTP systems whose main focus is not just obtaining the best performance possible, but also other objectives such as maintainability, support for ad-hoc transactions, etc.

Results analysis. Figure 3.7 shows that staged compilation of transaction programs produces C++ code that is twice as efficient as the hand-optimized C++ code written by experts. According to Table 3.1, not only are some optimizations missed by the experts, but also Beta applied the optimizations more thoroughly. In addition, Figure 3.7 shows that an optimized TPC-C implementation using Beta in a high-level language (i.e., Scala) is not more than twice

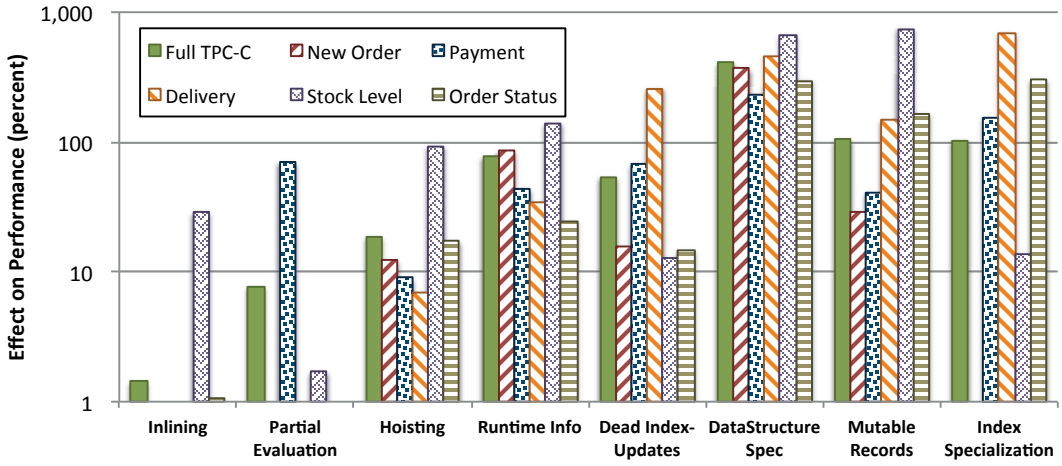


Figure 3.8 – The impact of important optimizations applied by Beta to TPC-C.

as inefficient as its C++ counterpart. Thus, Beta optimizations are more relevant than its target language. The figure also shows a performance degradation with an increase in the number of warehouses. This happens due to the cache misses associated with processing data from different warehouses. However, the C++ implementation of [109] does not experience this as their B+Tree Indexes are more cache insensitive compared to our Hash Indexes.

The breakdown of the impact of individual optimizations for TPC-C is shown in Figure 3.8. In addition, we also show the breakdown for each of the individual transactions. Naturally, the impact of different optimizations differs across transactions according to their characteristics. First, let us consider the `Removing Dead Index Updates` optimization. In the TPC-C benchmark, `Delivery`, `NewOrder`, and `Payment` transactions contain update operations to non-key columns. All of these updates are dead if the `Mutable Records` optimization is enabled. Figure 3.9 confirms the correlation between the impact of `Dead Index-Updates` and the fraction of time spent in update operations in each of these transactions in the absence of this optimization. If f_u is the fraction of update operations, after plugging in ∞ for s_u, we get the theoretical impact of the optimization to be $(1 - f_u)^{-1}$.

Next, we have `Index Specialization` (see section 3.4.2) where Beta introduces min-index for the `NewOrder` table, max-index for the `Order` table, and median-index for the `customer` table. Transactions `Delivery`, `OrderStatus` and `Payment` use these indexes for efficiently performing their operations. Figure 3.10 shows the composition of these transactions before and after applying this optimization. We can see that in `Delivery`, the operations to find the oldest `NewOrder` constitute roughly 89% of the transaction. Introducing a min-index speeds up this block by a factor of 290, thus achieving an overall speed up of around 800%. In `Payment`, finding the median `customer` takes roughly 67%. With the median-index, it is 8x as fast, giving an overall improvement of 143% to the transaction. `OrderStatus` contains two blocks, one finding the median `customer` by name (74%) and the other finding the latest `Order` (13%).

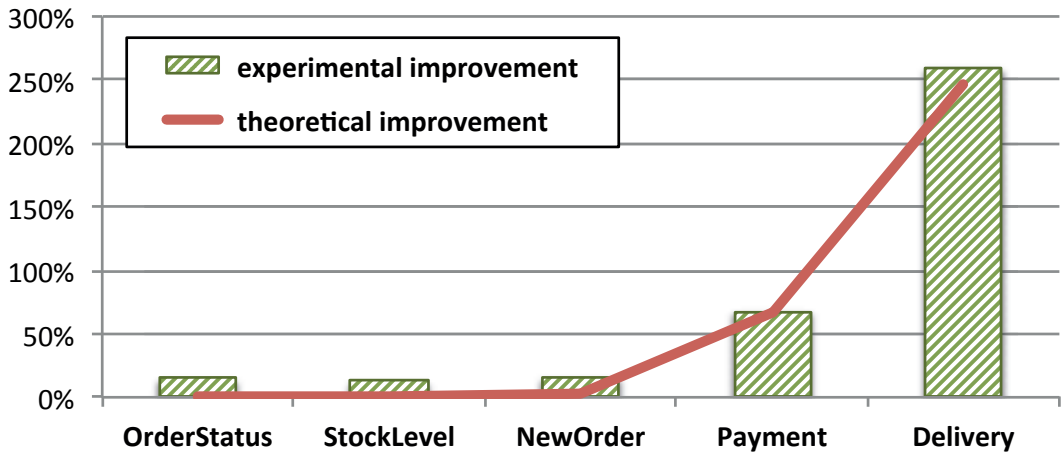


Figure 3.9 – Correlation between theoretical and experimental performance improvement by Dead Index Update on TPC-C transactions. The theoretical improvement is computed based on the fraction of updates in each transaction.

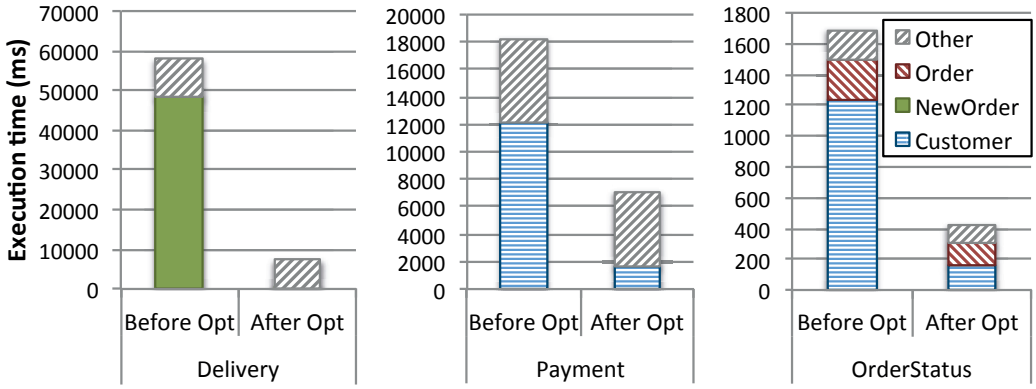


Figure 3.10 – Breakdown of the amount of time spent on operations of each table for TPC-C transactions with minimum, maximum and median operations, before and after applying `Index Specialization`.

By introducing the median and max indexes, respectively for these blocks, we get a speedup factor of 8 and 1.5, respectively, contributing to an overall speedup of 250%.

Among all transactions, `StockLevel` creates the maximum number of temporary objects with 200 `stock` records to be used as temporary keys for primary key lookup. These `stock` records comprise many string columns and are relatively more expensive to construct. Their construction and destruction form a significant portion of the `StockLevel` transaction, and therefore, when `Hoisting` moves these temporary objects out, this transaction has the highest impact.

Without `Partial Evaluation`, about 50% and 20% of the `Payment` transaction are taken up by the two string formatting functions (*sprintf*). However, this optimization evaluates the format string and converts these operations into efficient string manipulations, speeding them up by 3x and 2x respectively, giving an overall speedup of 75% to the `Payment` transaction.

With `Runtime Information` (see section 3.2), Beta infers the approximate number of records in all tables, and pre-allocates memory for index structures. This avoids costly index resizing at runtime. Furthermore, it also infers that no records are inserted to or deleted from the `warehouse`, `district`, `item` and `customer` tables after the data loading phase. This enables Beta to generate array indexes for these tables with efficient mapping from key columns to array positions that guarantee constant time lookup.

3.6.3 View Maintenance Triggers for TPC-H

We see DBToaster [1, 62] as a generator for a class of trigger programs: the DBToaster front-end turns classical complex queries into triggers of multiple smaller-step materialized view updates. We automatically rewrite such programs in Beta DSL. Our main transformation consists of less than 70 lines of Scala.

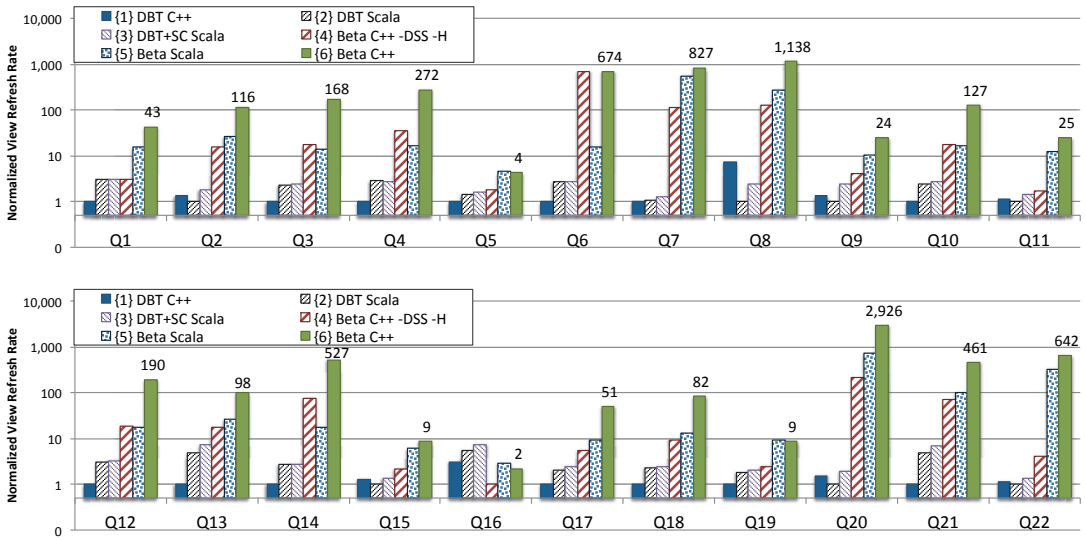


Figure 3.11 – Normalized view refresh rate for IVM triggers in TPC-H queries.

We compare the Beta DSL with the original DBToaster back-end (M3 to Scala). Our compiler is shorter to write (Table 3.3) and produces more efficient code. To show its efficiency, we compare the original DBToaster back-end with our compiler. In this benchmark, we measure the average throughput (during 1 minute) of *incremental view maintenance* (IVM) on a 200MB TPC-H dataset. All the tables are initially empty, and the task is to compute the query result after receiving a new tuple for each of the base relations present in the query. The order of inserting the tuples is consistent among different experiments. We do not compare with other systems as none implements similarly aggressive incrementalization techniques. In Figure 3.11 we compare different compiler implementations for TPC-H trigger programs:

- {1}*DBT C++*: original DBToaster C++ compiler.
- {2}*DBT Scala*: original DBToaster Scala compiler.
- {3}*DBT+SC Scala*: we modified the low-level Scala code generated by DBToaster to use the generic optimizations of the SC framework (DCE, CSE, and code motion).
- {4}*Beta C++ -DSS -H*: C++ program generated by Beta after disabling `Data Structure Specialization` and `Hoisting`.
- {5, 6}*Beta Scala/C++*: the program resulting from staged compilation. The input transaction code for Beta is similar to *DBT Scala*, except that it uses Beta DSL instead of standard Scala collections. The performance of the compiled programs for both Scala and C++ output languages are measured.

It is worth mentioning that even the baseline is fast and outperforms other IVM techniques [1]. The values shown in Figure 3.11 are normalized view refresh-rate and not the absolute throughput value. In all cases, we normalized other values with respect to the minimum value.

Results analysis. From the results in Figure 3.11, we make the following observations: By comparing {2} (*DBT Scala*) and {3} (*DBT+SC Scala*), we see that low-level optimization techniques (DCE, CSE, code motion and limited deforestation) give a small performance improvement to

Table 3.3 – Compiler implementation statistics in lines of code.

Component	DBToaster	Beta	Ratio
SQL incrementalization*	15,874	–	–
M3 to Scala compiler	14,660	1,561	11%
SC ** and inlining***	1,891	2,713	143%
Runtime libraries	2,150	5,851	272%

* This is the codebase for converting a SQL query into its corresponding incremental view maintenance trigger programs in an intermediate language, named M3.

** Does not include SC infrastructure (27,433 lines)

*** Available only in the new compiler

DBToaster programs (at most 1.4× and on average 36% speedup for 22 TPC-H queries). On the other hand, the programs compiled using Beta gained more than three orders of magnitude on some queries by applying all the optimizations and an overall 254× average speedup. This much is the difference between only using generic compiler optimizations and staged compilation of database application programs using Beta.

The optimizations in Beta are modular and can be turned on and off independently of each other. This allows us to measure the individual impact of each optimization. {4} (*Beta C++ -DSS -H*) shows the performance of Beta in absence of the most influential optimizations and the aggregated break-down of the impact of important optimizations is shown in Figure 3.12. In this figure, we investigate the median, average and maximum impact of these optimizations. We do not show *automatic index introduction* for generated IVM programs as DBToaster already enables this feature by default.

As shown in Figure 3.12, `Data Structure Specialization` has the most impact on this benchmark. `Q22` has the largest number of index lookups and field accesses to the data records. Thus, avoiding indirections for these accesses has a tremendous impact on its performance. The impact of `Partial Evaluation` can be attributed to partially evaluating the regular expressions used in the queries and moving them out of the critical execution path. The most significant case is that of `Q13` where there is regular expression matching for each record that is inserted into the `Order` table, which has the highest number of rows in the dataset. In the case of `Dead-Index Updates`, `Q1` shows the maximum improvement owing to its many update operations. Applying the `Mutable Records` optimization impacts `Q11` most, as it has the largest number of index lookups and alterations combined.

Figure 3.12 illustrates that employing the `Runtime Info` does not have a great impact, as the trigger programs mainly use small-sized hash-tables that do not require resizing frequently. Thus, using this optimization, which avoids resizing indexes, does not have a substantial performance improvement. In addition, as hash-indexes are the only type of index used for these trigger programs, index updates are relatively cheap and `Removing Dead Index-Updates` has a moderate impact.

The view maintenance programs for TPC-H queries have many function composition opportunities and thus applying deforestation and `CPS` has a high impact as shown in Figure 3.12.

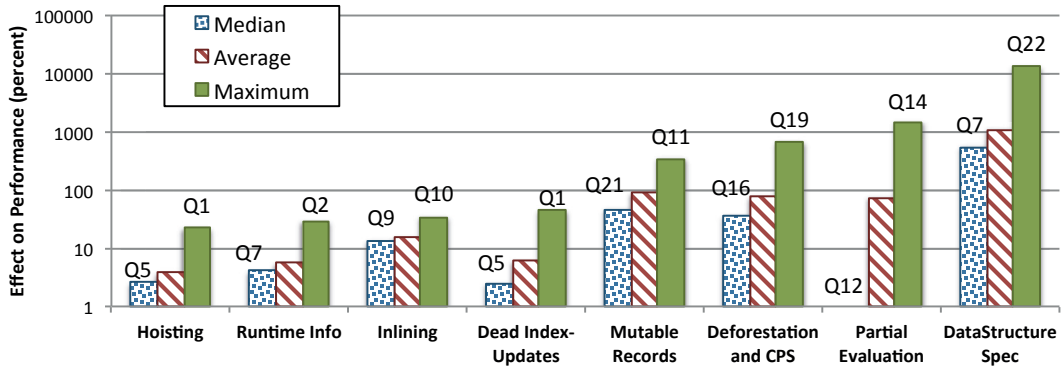


Figure 3.12 – Median, average and maximum impact of important optimizations applied by Beta to incremental view-maintenance triggers generated by DBToaster for TPC-H queries. The queries with maximum and median impact are shown as data labels on the graph.

This optimization has the most impact on `Q19` as it has a chain of seven function compositions in the critical path of the program.

3.7 Concurrency Control

In the previous sections, we only focused on achieving the best performance on a single CPU core. In this section, we discuss about the implications of having concurrency control algorithms for Beta, when a shared-nothing model is not completely satisfactory or applicable.

3.7.1 Concurrency Model

The shared-nothing model with partition-level timestamp ordering, which is the concurrency model used in H-Store [54], is the best match for applying aggressive optimizations to database application programs, as all the programs on a partition are sequentially handled by an isolated worker thread, without any data sharing with other threads. This assumption gives more flexibility to Beta to apply the optimizations that are only correct in the case of having a single thread that accesses and manipulates the underlying data structures. Beta can be directly used as a staging compiler for stored procedures at the heart of H-Store.

Even though the H-Store concurrency model performs best for the partitionable workloads [54], not all workloads are perfectly partitionable. The concurrency control algorithms that operate in a shared-memory model (e.g., multi-version concurrency control (MVCC), optimistic concurrency control (OCC), two-phase locking (2PL)), are a better fit for the latter workloads [32, 120]. Using these mechanisms with a shared-memory model has two implications on Beta: 1) some optimizations are not applicable as-is when these concurrency control algorithms are used, and 2) there might be opportunities for having new optimizations that are tailored for a specific concurrency control algorithm.

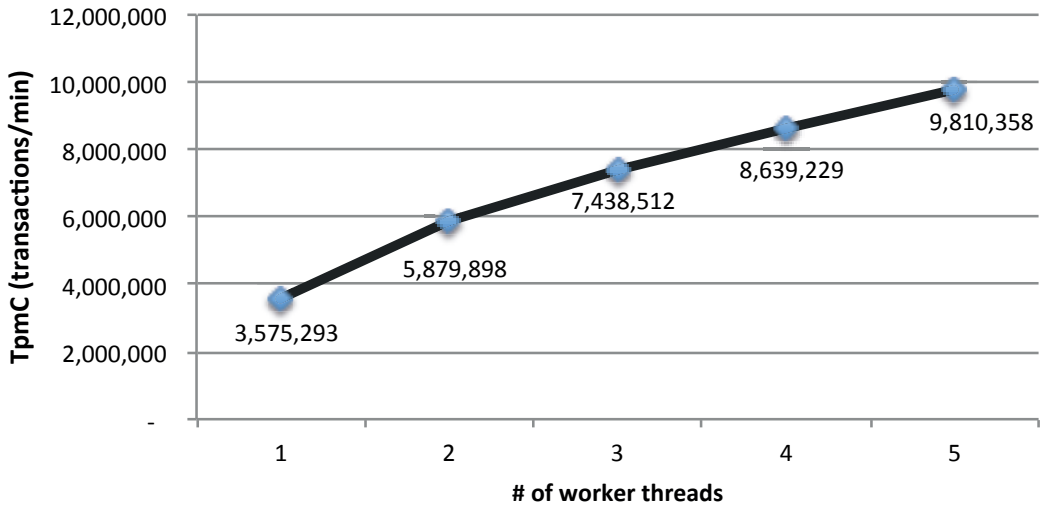


Figure 3.13 – The scalability of the TPC-C benchmark results using the most optimized generated code by Beta and run under the shared-nothing model (similar to H-Store).

3.7.2 Impact on Optimizations

All the optimizations that we have described for a single-threaded scenario in section 3.4 are still applicable when multiple threads run on partitioned data. However, as using a shared-memory model requires a corresponding change in the underlying data structures, the optimizations that are tied to the data structure have to be redefined. For example, using MVCC implies maintaining several versions for the base data and the indexes are handled accordingly, e.g., by having multi-versioned indexes. In this case, the definition for the `Mutable Records` optimization changes. In a multi-version environment, unlike the single-version scenario, it is necessary to copy-on-write and create a new version upon each update. In order for `Mutable Records` optimization to work in this setting, one naive approach is to have `get` operations that always return a new version of the record on each call. However, this approach is sub-optimal, as not all read operations are followed by an update. Instead, Beta first performs an analysis to identify the `get` operations that are followed by an `update` and transforms only those `get` operations to `getForUpdate` operations, which returns a newly created version upon each call. Then, any update operation is directly applied on this version, as it is created for this update.

3.7.3 Experiments

Next, we illustrate the performance of the TPC-C benchmark compiled using Beta, both when it uses shared-nothing and shared-memory models. These experiments show that using program analysis and compilation techniques still have a significant impact on the performance improvement of database application programs, even in the presence of concurrent and parallel execution.

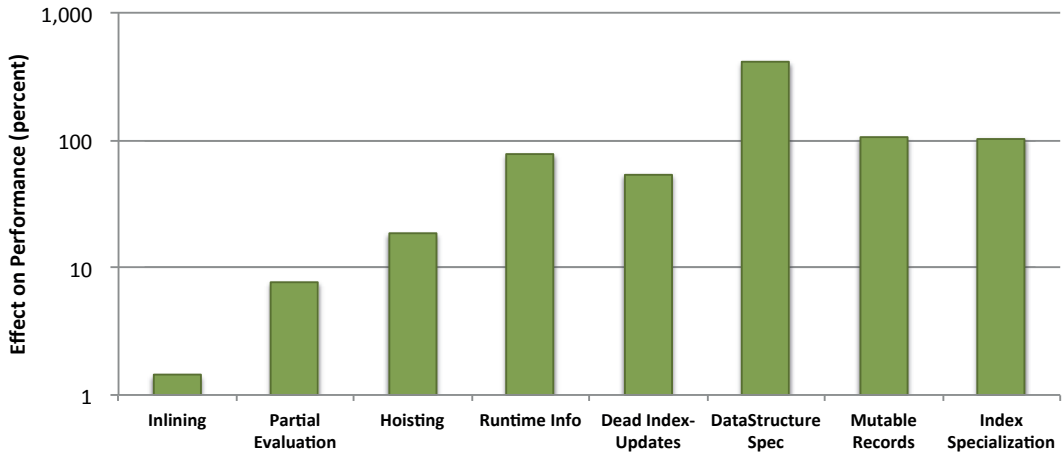


Figure 3.14 – The impact of important optimizations applied by Beta to TPC-C and ran under the shared-nothing model (similar to H-Store) with five worker threads.

In order to run the TPC-C benchmark in a shared-nothing model, we took the approach of [109]. Even though the programs in TPC-C are not perfectly partitionable, it is still possible to divide the programs into smaller pieces, where each piece is executed on one partition, without requiring any synchronization between partitions. This requires a global ordering on the programs executing across partitions, to make sure that the execution is serializable [114].

Figure 3.13 shows the overall throughput of TPC-C on the most optimized code generated by Beta. As expected, the optimizations by Beta do not hurt the scalability of this benchmark. In addition, we measured the impact of individual optimizations on the performance of the TPC-C benchmark, when it is run in the shared-nothing model with five worker threads. Again, as expected, the impact of each individual optimization completely matches the one for the single-thread benchmarks presented in section 3.6.2.

For the shared-memory model experiments, we chose optimistic MVCC [84] as the concurrency control algorithm. Then we generated the most optimized programs using Beta, after applying the necessary modifications to the optimizations for this algorithm. Figure 3.15 shows the throughput for TPC-C in the shared-memory model with different threads for different combination of optimizations. Our code scales linearly with the increase in the number of threads in each of the different optimization combinations, although not at the same rate. The most optimized version not only has the best performance, but also scales the best. The figure also shows that both `Record Structure Specialization` as well as `Specialized Indexes` optimizations play a very important role in achieving good performance in the shared-memory model as well. On the other hand, even though other optimizations such as `Hoisting` and `Partial Evaluation` still optimize the code, their impact is much less compared to the single-threaded model. This is because the performance of the single-threaded model is much better than the base performance (i.e, one thread) of the shared-memory model. As a result, the blocks of code improved by these optimizations form a higher fraction (f_i) of the execution time in the generated program (before the optimization) in the single-

threaded model as compared to the shared-memory model, making their impact in the former scenario higher.

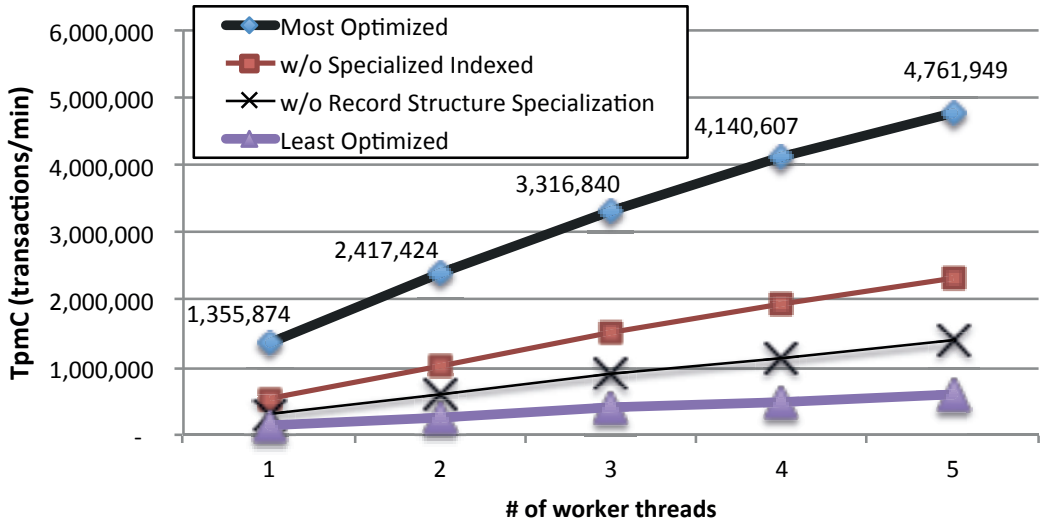


Figure 3.15 – The scalability of the TPC-C benchmark results using the generated code by Beta (with different optimizations) and run under optimistic MVCC.

3.7.4 Improving Concurrency Control Algorithms

In this section, we showed the effectiveness of Beta in presence of using concurrency control algorithms. In addition, there are certain features of the concurrency control algorithms that benefit from the automated program analysis. Consider the case of the attribute-level validation in optimistic MVCC (OMVCC) [84] instead of the record-level. To do fine-grained attribute-level validation, OMVCC needs to know about the accessed and modified columns for each record. This information can be expensively gathered at runtime, but a better approach is to precompute these columns during compilation. In addition, there are further optimizations that reorder operations to follow the best-practices while using the concurrency control algorithm. As an example, when using 2PL, the access and modification operations of the high-contention records are moved closer to the end of the program in order to reduce the lock lifespan. Furthermore, in the next chapter, we propose a new variant of the MVCC algorithm and we show how program dependency analysis is employed by the algorithm to improve its performance.

4 Transaction Repair for Multi-Version Concurrency Control

4.1 Introduction

Recent research proposes an optimistic MVCC algorithm as the best fit for concurrency control in in-memory databases [84]. This algorithm, like its predecessors [71, 54], aborts and restarts the conflicting transactions, which is simple but sub-optimal. Any conflict among transactions results in more work for the concurrency control algorithm and an increase in the execution latency of the transactions. Increased latency not only affects the throughput of individual transactions, but also increases the probability of having more concurrent transactions in the future. This might incur even more conflicts, forming a negative feedback loop.

The sub-optimality of the abort and restart approach becomes more significant when the number of conflicting transactions is high. The two factors that contribute the most to an increase in the number of conflicts are: (1) having high contention data objects that are read and updated by several concurrent transactions, and (2) having long running transactions, the lifespan of which intersects that of many other transactions.

In this chapter, we introduce a novel multi-version timestamp ordering concurrency control algorithm, called Multi-Version Concurrency Control with Closures (MV3C). This algorithm resolves the conflicts among concurrent transactions by only partially aborting and restarting them. The main challenges for this type of conflict resolution technique are having: (1) low overhead on the normal execution of transactions, as every transaction pays this cost irrespective of encountering a conflict, and (2) a fast mechanism to narrow down the conflicting portions of the transactions and fixing them, as a mechanism that is slower than the abort and restart approach defeats the purpose. The first challenge is dealt with by reducing additional bookkeeping by reusing the existing concurrency control machinery. To address the second challenge, MV3C uses lightweight dependency declaration constructs in the transaction programs that help in immediately identifying the dependencies among different operations. Using this dependency information, MV3C pinpoints blocks of the program affected by the conflicts and quickly re-executes only those blocks. The dependency information is added either manually by the user, or by employing static program analysis and restructuring.

The rationale for proposing MV3C is that a conflict happens only if a transaction reads some data objects from the database that become stale by the time it tries to commit. Here, the assumption is that all data modifications made by a transaction are invisible to other transac-

tions during its execution. These modifications become visible only after the critical section during which the transaction commits. Consequently, just before committing a transaction, by checking whether its data lookup operations read the most recent (committed) versions of the data objects, serializability of the execution is ensured. In addition, by associating read operations with the blocks of code that depend on them, the portion of a transaction that should be re-executed in the case of a conflict is identified quickly, and gets re-executed. These blocks correspond to a specific class of sub-transactions in the nested transaction model. The boundaries of these blocks are specified by MV3C, which makes it possible to efficiently repair conflicting transactions. This is further discussed in section 5.6.

Our experimental results show that MV3C can achieve more than an order of magnitude increase in transaction processing throughput for high-contention scenarios compared to optimistic MVCC. Specifically, under high-contention, MV3C achieves around twice the throughput of optimistic MVCC for the well-known TATP[130] and TPC-C[118] benchmarks. In particular, this chapter makes the following three contributions:

1. An efficient conflict resolution mechanism for multi-version databases, MV3C, which repairs conflicts instead of aborting transactions, with minimal execution overhead. The design of this mechanism is discussed in section 4.2.
2. A method to deal with write-write conflicts in MV3C. Unlike other optimistic MVCC algorithms, MV3C can avoid aborting the transaction prematurely when a write-write conflict is detected. This is discussed further in section 4.2.3.
3. A mechanism for fixing the result-set of failed queries for MV3C, which can optionally be enabled for each query in a transaction. This mechanism can boost the repair process for transactions that have long running queries. The details of this mechanism are described in section 4.4.2.

Motivating examples. The three main cases of transaction programs that benefit from MV3C are illustrated in Figure 4.1. Each case in this figure shows an instance of a transaction program starting with a *begin* command and finishing with a *commit* command. Each program consists of one or more blocks of code represented by a box. A block of code consists of any valid sequence of commands. The dependencies among different blocks of code are represented by arrows. If block Y depends on block X, there is an arrow from X to Y. Moreover, it is assumed that each of these three instances failed during its validation phase, because of a conflict detected in its block A.

In the following, the three scenarios shown in Figure 4.1 are described using examples and the approach of MV3C for resolving the conflict is explained for each case.

First. In this case, the transaction has logically disjoint program paths and conflicts happen only in a few of them. Such paths are detected from the program structure and only they are re-executed. One example of this case is shown in Figure 4.1(a).

Example 4.1. Assume that in Figure 4.1(a), block A reads a row from table T_A and updates it, and block B reads a row from table T_B and updates it. Moreover, these updates only depend on the input parameters of the program. Then, if a concurrent transaction also updates the same row from table T_A and commits before the other transaction, the latter fails to commit.

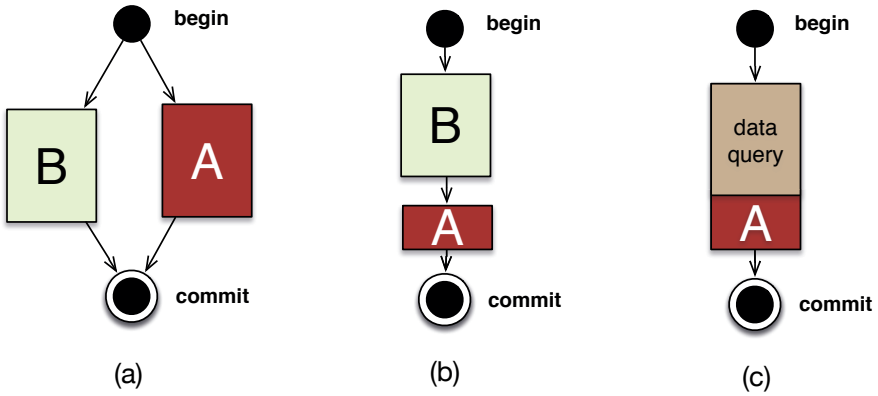


Figure 4.1 – Cases where MV3C is more efficient in repairing the conflicting transactions compared to the “abort and restart" approach.

However, MV3C detects that only block A has a conflict and re-executes only this block, without re-executing block B. △

Second. In this case, conflicts happen after doing a substantial amount of work in the transaction. Here, it is not necessary to redo all the work. Instead, the data available before the conflict is reused in order to continue from the point of conflict. Figure 4.1(b) shows this case, where only the conflicting block A is re-executed. A concrete example of this case is the Banking example described in Example 4.2. In the following sections, this example is used in order to better describe MV3C.

Example 4.2. Banking example: The example consists of a simplified banking database with an Account table. This table stores the balance of the customers identified by an ID. There are two types of transaction programs that run on this database. The first program, named SumAll, is read-only. A SumAll transaction sums up the balances in all the existing accounts. The second program is named TransferMoney and it is written in a PL/SQL-like language as shown in Figure 4.2. A TransferMoney transaction transfers a specific amount of money from one account to the other, given the availability of sufficient funds. The money transfer also consists of a fee that is deducted from the sender account and is added to the central fee account identified by FEE_ACC_ID.

Now, assume that two TransferMoney transactions, using different input parameters, run concurrently. Then, the first one succeeds, and the other one fails due to line 17 in Figure 4.2. MV3C detects that only the operation that reads the current value of the fee account impacts the correctness of line 17. Thus, only that line gets re-executed, this time, with the new value of the fee account. △

Third. In this case, conflicts occur in the beginning of the transaction. Then, the data returned by SELECT queries to the database is reused after accommodating the changes introduced by the conflicting transaction(s). Thus, the re-evaluation of queries from scratch is avoided. As illustrated in Figure 4.1(c), even though the transaction program consists of a single block of code, the initial part of the block responsible for querying the data from the database is re-executed more efficiently under MV3C.

Example 4.2 (continued)**.** In the Banking example, assume that there is another transaction program, named Bonus. This transaction program increments the balance of the accounts with a minimum balance of 500 by one. As the *balance* column is not indexed, a Bonus transaction has to scan the whole Account table. Meanwhile, a concurrent TransferMoney transaction commits, increasing the balance of an account above 500. Then, the Bonus transaction fails validation, as it did not consider the updated record in the Account table. However, as MV3C knows that the conflict happened only because of that record, it fixes the result-set of the query by including the additional record. This avoids another full scan over the Account table. △

4.2 MV3C Design

The main idea behind MV3C is that by exposing the program dependencies to the concurrency control algorithm, conflicts among concurrent transactions can be resolved efficiently. This information is already provided to the transaction processing system via user defined transaction programs [37, 115, 133]. However, an abstract view of the program is needed to exploit this dependency information. In this abstract view, correctness checks are associated with blocks of code. Then, each executed instance of the transaction program, referred to as a *transaction*, uses this information to recover efficiently from a failure due to a conflict.

For this purpose, the possible failure points are identified and the transaction program is partitioned into smaller blocks such that each failure is contained within a single block. These program blocks are independent, and the failure of some blocks does not affect the other blocks in any way. The failure possibility stems from having a predicate in that block of the program, which might not pass the validation phase. After a failed validation, a new timestamp is assigned to the transaction, but only those blocks that have an invalid predicate inside them are rolled back and re-executed. The validation semantics guarantees that the other predicates would return the same values as the initial execution, and therefore, re-executing them is unnecessary.

In the rest of this section, we describe the design of MV3C. MV3C builds its efficient conflict resolution mechanism on top of the algorithm proposed in [84]. Throughout the rest of this chapter, this algorithm is referred to as OMVCC, where O stands for *Optimistic*. A brief overview of OMVCC is provided before going into the details of MV3C, as the latter borrows some features from the former.

4.2.1 OMVCC Overview

OMVCC gathers predicates for all the read operations of a transaction. A predicate in OMVCC can be thought of as a logical condition created using the attributes of a relation, encapsulating a data selection criterion. For example, the WHERE clause in a SELECT statement on a single table is a predicate for that table. The candidate predicates in our example program are highlighted in Figure 4.2. Moreover, OMVCC makes a reasonable assumption that every transaction writes into only a limited number of data objects. Therefore, it is feasible to keep track of the write-set of a transaction (i.e., the *undo buffer*) during its execution.

```
1 /* fm = from, acc = account and bal = balance */
2 TransferMoney(fm_acc, to_acc, amount) {
3  START;
4  SELECT bal INTO :fm_bal FROM Account WHERE id=:fm_acc;
5
6  IF(amount < 100) fee = 1.0;
7  ELSE fee = amount * 0.01;
8
9  IF(fm_bal > amount+fee) {
10   SELECT bal INTO :to_bal FROM Account WHERE id=:to_acc;
11
12   fm_bal_final = fm_bal - (amount + fee);
13   to_bal_final = to_bal + amount;
14
15   UPDATE Account SET bal=:fm_bal_final WHERE id=:fm_acc;
16   UPDATE Account SET bal=:to_bal_final WHERE id=:to_acc;
17   UPDATE Account SET bal=bal+:fee WHERE id=:FEE_ACC_ID;
18   COMMIT;
19  } ELSE ROLLBACK;
20 }
```

Figure 4.2 – TransferMoney transaction program from the Banking example (Example 4.2) in a PL/SQL-like language.

As OMVCC is an optimistic algorithm, it requires a validation phase before a successful commit. A variant of *precision locking* [57] is used for validating transactions. This approach requires that the result-sets of all the read operations are still valid at commit attempt time, as if the operations were done at that time. This creates an illusion that the whole transaction executed at commit time.

To achieve this, all committed transactions are stored in a list called *recently committed* transactions. During the validation phase of a transaction T, all of its predicates are checked against all committed versions of the transactions in the recently committed list. From this list, only those transactions that committed during the lifetime of T are considered. The undo buffers of these (concurrently executed) transactions contain the changes of this time period. If any committed version satisfies one of T's predicates, then the data read by the transaction is obsolete. The newer version should have been read instead, if the read operation used the commit timestamp as its reference point. In this case, T fails validation, rolls back and restarts.

4.2.2 MV3C Machinery

MV3C gathers the predicates used during the execution of a transaction, similar to OMVCC. Moreover, MV3C creates a new version for each modification of a data object. Each data object keeps a list of versions belonging to it, called its *version chain*. When a version is created for a data object, it is added to the head of this chain. The notion of *version* is defined below.

Definition 4.1. *A **version** is a 4-tuple* (T, O, A, N)*, where* T *is either the commit timestamp of the transaction that created the version, or the transaction ID if the transaction is not committed,* O *is the identifier of the associated data object,* A *is the value of* O *maintained in this version, and* N *is a version identifier if more than one version is written by* T *for* O*.*

The value written in a version is immutable. It cannot be modified, even by its owner transaction. Given a version V, the version identifier N in V is used to differentiate distinct versions written by the same transaction for the same data object. However, after a transaction is committed, only the newest version written by the transaction becomes visible to the other transactions. In practice, N can be a pair of pointers, pointing to the older and newer versions in the internal chain of versions written by a single transaction. Then, the committed version is the one without a newer version in N. The notion of a committed version is defined below.

Definition 4.2. ***Committed version:*** *a triple* (T, O, A)*, where* (T, O, A, N) *is a version such that* N *is the identifier of the latest version of* O *written by the transaction with commit timestamp* T*.*

The read operations in MV3C traverse the version chain until a visible version is reached. A visible version either is owned by the transaction itself or is the latest committed version before the transaction started.

Definition 4.3. ***Visible version:*** *a version* (T_1, O, A, N_1) *is visible to a transaction with start timestamp* T_2 *if either:*

- T_1 *is committed,* $T_1 < T_2$ *and there is no other version* $(T_3, O, \_, \_)$ *where* $T_1 < T_3 < T_2$*, or*
- $T_1 = T_2$ *and there is no other version* $(T_1, O, \_, N_2)$ *where* N_2 *is newer than* N_1*.*

MV3C transactions, like those in OMVCC, have an undo buffer that maintains the list of versions created by the transaction. After a transaction commits, its undo buffer contains only the committed versions. The undo buffer is a representative of the effects of a committed transaction, as it contains all the modifications done by the transaction to the multi-version database. In addition, each predicate has a closure bound to it. The term *closure* is used in the sense of the programming languages literature [70]. In the rest of this chapter, the term *predicate* refers to an *MV3C predicate*, unless otherwise stated.

Definition 4.4. *An **MV3C predicate** X consists of (1) a data selection criterion, (2) a closure* $C(X)$ *bound to it, (3) a list of versions* $V(X)$ *registered to it, and (4) a list of child predicates* $D(X)$*.* $V(X)$ *and* $D(X)$ *are populated after* $C(X)$ *is executed.*

Definition 4.5. **C(X)** ***(the closure bound to predicate* X*)*** *is a deterministic function that encloses all operations in a transaction program that depend on the result of* X*.* $C(X)$ *receives the result of evaluating* X *along with a set of context variables as its parameters.*

Example 4.2 (continued)**.** The equivalent program of Figure 4.2 translated into MV3C DSL is illustrated in Figure 4.3. In Figure 4.3, the shaded part of P_1 is an MV3C predicate whose closure is the gray box underneath it. △

A closure can contain data selection or data manipulation operations in addition to computations. Each data selection operation creates a new predicate with its own closure, which

has access to its parent predicates and their result-sets. Moreover, each data manipulation operation (i.e., insert, delete and update statement) creates a new version of the data object.

MV3C requires specifying the relationships of the predicates to the versions and to the other predicates. A new Domain Specific Language (DSL), called MV3C DSL, is used in MV3C for writing transaction programs. MV3C DSL is meant to capture dependencies among operations inside a transaction program. Basically, it is a simple library for encoding the relationships among the predicates, and binding closures to them. It also defines the granularity in which the algorithm operates, which is at the statement level by default. A naïve pessimistic translation of an existing program written in a PL/SQL-like language to the MV3C DSL requires a simple dependency analysis. One such translation can be derived by assuming that each operation in the transaction program depends on all of its previous operations. The same naïve approach can be used to execute ad-hoc transactions, where the whole transaction is not given to the system in advance, i.e., the ad-hoc commands of a transaction are issued from an application program. Program restructuring can be used along with detailed program analysis to generate more accurate dependency constructs. Further details about translating transaction programs written in a PL/SQL-like language to the MV3C DSL are described in section 4.8.

```
/* fm = from, acc = account and bal = balance */
TransferMoney(fm_acc, to_acc, amount) {
 START;

  IF(amount < 100) fee = 1.0;
  ELSE fee = amount * 0.01;

 P1 Account WHERE id=:fm_acc => fm_acc_entry
 IF(fm_acc_entry.bal > amount+fee) {
  fm_acc_entry.bal -= (amount + fee);
  fm_acc_entry.persist();

  P2 Account WHERE id=:to_acc => to_acc_entry
  to_acc_entry.bal += amount;
  to_acc_entry.persist();

  P3 Account WHERE id=:FEE_ACC_ID => fee_acc_entry
  fee_acc_entry.bal += fee;
  fee_acc_entry.persist();
  COMMIT;
 } ELSE ROLLBACK;
}
```

Figure 4.3 – An example transaction program in the MV3C DSL.

To specify the relationships among predicates, when a predicate is created in the closure $C(X)$ of some predicate X, it is added to $D(X)$, the list of child predicates of X. For example, in Figure 4.3, P_2 and P_3 are added as children of P_1. A graph is constructed internally during the execution, using this parent-child relationship information. By definition, the predicates form

a directed acyclic graph (DAG), because there cannot exist any edge from a new predicate to an older predicate and hence a cycle cannot be formed. More intuitively, a SELECT statement cannot be executed if it requires a parameter that is available only after executing another SELECT statement in the future. Consequently, the result of executing a transaction program in MV3C is a DAG of predicates with a closure assigned to each predicate. The DAG for a transaction T is called the *predicate graph of* T. In addition, to specify the relationships between predicates and versions, references to the newly created versions are stored in $V(X)$, the list of versions registered to X. Then, using these references, X keeps track of the created versions that directly depend on its result-set.

Apart from the data objects that are altered in the database, there can also be mutable variables in a transaction program. Each mutable variable is local to a closure, as sharing it among more than one closure makes it impossible to reason about the state of the variable if any of these closures is re-executed. However, an immutable copy of the variables defined in a closure is shared with the closures of its child predicates. If a mutable variable is shared and modified by different closures, it is treated as a database object and multi-versioning will be applied to it.

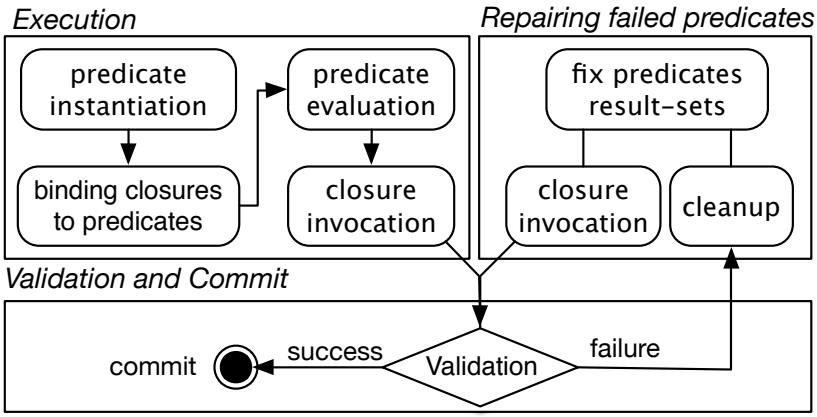


Figure 4.4 – The life cycle of a transaction running under MV3C.

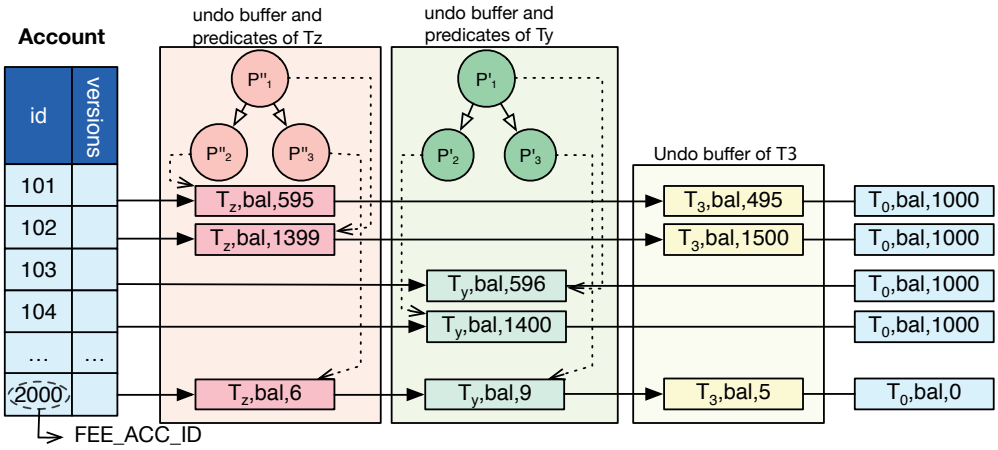
For $C(X)$, the context variables are those variables that are defined outside the closure and are accessible to it. The context variables are immutable and are categorized into: (1) the input parameters of the transaction, (2) the result-sets of the ancestor predicates of X, and (3) the variables defined in the closures bound to the ancestor predicates of X. For (2) and (3), MV3C guarantees that if either the result-sets of the ancestor predicates or the variables defined in their closures change, operations of $C(X)$ are undone.

Example 4.2 (continued)**.** In Figure 4.3, there are three predicates, P_1, P_2 and P_3, where P_2 and P_3 depend on P_1, as they are defined inside the closure of predicate P_1. Each predicate in this example has a closure that is represented as a gray box underneath it. The orange shade is the predicate itself, and the arrow symbol (=>) in front of each predicate represents the execution of the predicate, which returns the result of the evaluation (i.e., a copy of the latest visible version of the requested data object). Then, the returned version is stored in a variable, which is used inside the closure that is bound to the predicate. Inside the closure, the fields of the returned version are accessed and modified. In the case of a modification, a call to

the *persist* function is necessary in order to store the changes into a new version inside the database, and adding it to the undo buffer of the transaction, as well as the list of versions of the predicate. △

Predicates are first class citizens in MV3C. Various types of predicates can be defined, all of which implement the predicate interface comprising an *execute* function and a *match* operation. The execute function accepts a closure as its parameter and binds it to the predicate. This function evaluates the predicate by collecting the visible versions that satisfy the data selection criterion into the result-set of the predicate. Then, the closure is executed with the result-set and the set of immutable context variables as its parameters. The match operation of a predicate checks whether a given version satisfies the selection criterion. It is used during the validation phase of a transaction. The lifecycle of an MV3C transaction is shown in Figure 4.4. The details of the execution, validation and repair phases of a failed transaction are described in sections 4.2.3-4.2.5.

4.2.3 MV3C Execution



recently committed transactions

commitTime	action
T_3	TransferMoney(101,102,500)

active transactions

xactID	startTime	action
T_x	T_1	Read only: Sum all bal
T_y	T_4	TransferMoney(103,104,400)
T_z	T_5	TransferMoney(102,101,100)

Figure 4.5 – A snapshot of the MV3C database for the Banking example.

An MV3C program starts by instantiating the root predicates of the predicate graph and calling their execute functions. Each root predicate is evaluated and the closure bound to it is

executed. This instantiates the predicates defined in the closure, which are evaluated, and in turn, their closures are executed.

Example 4.2 (continued)**.** Figure 4.5 illustrates a snapshot of the Banking example run under MV3C. P_1'', P_2'' and P_3'' are the predicates used in T_z. They are the runtime instances of predicates P_1, P_2 and P_3 shown in Figure 4.3. The dotted arrows represent references to their corresponding created versions. △

A transaction finishes its execution when the calls to the execute functions of all the root predicates return successfully. However, some transactions may abort prematurely, i.e., before reaching to the end of execution. One reason is a transaction issuing an abort command after detecting an unwanted state. In this case, repairing or restarting the transaction is irrelevant. The rollback is performed and all the versions created until that point are discarded. Another reason for a premature abort is having a write-write conflict, described next.

Handling multiple uncommitted versions

A transaction can be involved in one or more read-write, write-read or write-write conflicts. Among these conflict types, optimistic MVCC algorithms deal with read-write and write-read conflicts during the validation phase. However, handling a write-write conflict is different. A write-write conflict happens when a transaction T tries to write into a data object that has another uncommitted version or a committed version that is newer than T's start timestamp. This type of conflict results in a guaranteed failure during the validation phase of MVCC algorithms such as OMVCC. As these algorithms do not have a method to recover from such conflicts, their best choice is to detect the write-write conflicts during the execution, abort the conflicting transaction prematurely and restart it.

Example 4.3. Consider a long running transaction that runs successfully until the last statement. The last statement writes the results into a highly contended data object. Rolling back the transaction just because of a conflict in this last statement might not be the best decision. On the other hand, if a write-write conflict is encountered in the first few statements of the program, and the rest of the program depends on the written data, then continuing the execution from that point would be a complete waste of resources. Here, the repair action to recover from this conflict is similar to restarting it from scratch. A conflict that happens in predicate P_3 shown in Figure 4.3 is of the former type, while one in predicate P_1 is of the latter type. △

For this reason, it is desirable to provide a configuration option for indicating how to handle write-write conflicts. The first option is to deal with a write-write conflict upon its detection by aborting and restarting the transaction early, as in [84]. If the programmer chooses this option, then the transaction involved in a write-write conflict is rolled back and restarted from scratch. Even though MV3C might still be able to save the work for the closures that are completely done before reaching the conflict, some additional bookkeeping is required to make it feasible. To minimize the overhead of MV3C, we deliberately abort and restart the transaction in this case as the programmer indicates. Operations that modify the key or indexed fields (e.g., insert or delete) are always treated this way and fail-fast if another uncommitted version exists in the version chain.

The second option to deal with write-write conflicts is to ignore the detection of the conflict and proceed forward with creating a new version of the data object. Under MV3C, if multiple transactions are allowed to write into a single data object, each creates its own version and adds it to the version chain. Before committing any transaction that is involved in a write-write conflict, these additional versions are not important, as they are visible only to their own transactions. The approach taken in the validation and commit phase to handle this case without affecting serializability is described in section 4.2.4.

Example 4.2 (continued)**.** In the snapshot shown in Figure 4.5, both T_z and T_y have concurrently written new versions for FEE_ACC_ID, because write-write conflicts were allowed in TransferMoney. Otherwise, if T_z executes the closure bound to P_3'' after T_y finishes all of its operations, T_z prematurely aborts and restarts, because of the uncommitted version written by T_y. △

We strongly believe that if there exists a method for conflict resolution of transactions (such as MV3C), premature abort and restart is not the best available option. Accordingly, the decision regarding this case should be made either by the transaction programmer or an automated program analyzer. In MV3C, the configuration of the write-write conflict behavior can be done as a system-wide or table-wide setting that can be overridden for each individual update operation.

4.2.4 MV3C Validation and Commit

A transaction T that is not aborted prematurely has to be validated before it can be committed. The validity of T is dependent on the validity of its predicates. Other transactions that committed during the execution of T are potentially conflicting with it. Hence, it is necessary to check if any of the potentially conflicting transactions write into a data object that T read.

Definition 4.6. ***Valid predicate:*** *A predicate P belonging to transaction T with start timestamp S is defined to be valid at timestamp S' if and only if the result-set of P is the same when S' is used as the start timestamp of T instead.*

Proposition 4.1. *A predicate P belonging to transaction T is valid at timestamp C if there is no new version committed between S and C that satisfies P, where S is the start timestamp of T.*

PROOF SKETCH. The predicate P is evaluated on the snapshot of the database at S. The newly committed versions between S and C exist in the undo buffer of the transactions committed in the same time period. These are the only changes that happened to the database in the same period. If P does not match any newly committed version, then there cannot be a change in its result-set. Thus, based on Definition 4.6, P is still valid at C. □

Accordingly, the validation phase consists of matching the predicates of the current transaction against the committed versions of potentially conflicting transactions. If none of the predicates match against any version, then the validation is successful, and the transaction commits. Otherwise, the matched predicates, along with all their descendant predicates, are marked as invalid. For this reason, the predicate graph is traversed in topological sort order, first

Algorithm 1 Validation

1: **Input**: $G(S) \leftarrow$ predicate graph resulting from executing a transaction program T using start timestamp S
$S' \leftarrow$ validation timestamp, where $S < S'$
2: **Output**: $L_1 \leftarrow$ list of valid nodes in $G(S)$ using S'
$L_2 \leftarrow$ list of invalid nodes in $G(S)$ using S' along with their descendants
3: $Q \leftarrow$ the result of applying topological sort on $G(S)$
4: **while** Q is non-empty **do**
5: $n \leftarrow$remove the node from the head of Q
6: **if** ((a parent predicate of n exists in L_2) or (n is an invalid predicate at S')) **then** add n to tail of L_2
7: **else** add n to tail of L_1
8: **end while**
9: **return** (L_1, L_2)

traversing the higher-level predicates. The Validation algorithm (Algorithm 1) is used by MV3C to validate transactions.

The Validation algorithm traverses the predicate graph to identify all invalid predicates and their descendants. This is different from OMVCC where the validation process stops after finding the first invalid predicate, as it cannot succeed anymore. By contrast, in MV3C the validation phase is responsible for identifying all invalid predicates. For this reason, the validation process resumes at the next predicate even though it is known that the transaction cannot succeed validation. This algorithm not only identifies the invalid predicates, but also topologically sorts and splits the predicate graph into two parts, L_1 and L_2. All valid predicates are in L_1, whereas the invalid predicates and their descendants are in L_2. The validation of a transaction, starting at timestamp S and resulting in a predicate graph $G(S)$, is successful at timestamp S', if L_2 in the result of Validation ($G(S)$,S') is empty, in which case the transaction commits with commit timestamp S'.

Example 4.2 (continued)**.** In the snapshot shown in Figure 4.5, assume that T_y finishes execution first. Then, as there is no concurrent transaction that committed before T_y, it passes validation, its commit timestamp is assigned, and it is added to the recently committed list as T_6. Thereafter, T_z enters the validation phase and checks the recently committed list and finds T_6 as a concurrent transaction that committed after it had started its execution. Using the Validation algorithm, T_z traverses the predicate graph in topological sort order and matches each predicate against the committed versions of T_6. The Validation algorithm finds a match for P_3'' against $(T_6, bal, 9)$, making the predicate invalid. P_3'' is the only predicate in this example that gets added to L_2, the list of invalid predicates. △

Lemma 4.1. *L_1 in the result of the algorithm Validation ($G(S)$,S') (Algorithm 1) does not contain any invalid predicate nodes or the descendants of an invalid predicate node, for any given timestamps S and S', and predicate graph $G(S)$, where $S < S'$.*

PROOF SKETCH. This follows immediately from lines 6 and 7 of the Validation algorithm. □

Lemma 4.2. *$L_1 + L_2$, the concatenation of L_1 followed by L_2, is a topological sort of $G(S)$, where L_1 and L_2 are the results of the algorithm Validation ($G(S)$,S') (Algorithm 1).*

PROOF SKETCH. Algorithm 1 starts by topologically sorting the predicate graph. It then divides the result of the topological sort into two lists, L_1 and L_2. Since any subset of a topological sort is also topologically sorted, L_1 and L_2 individually follow the topological sort. To prove that $L_1 + L_2$ is topologically sorted, it is sufficient to show that no node in L_2 has a descendant node in L_1. In Algorithm 1, the only place where nodes are added to L_2 is line 6. Based on the condition in line 6 and the fact that nodes are traversed in topological sort order, if a node is in L_2, all of its descendant nodes are also in L_2. □

Impact of multiple uncommitted versions on validation

As explained in section 4.2.3, if write-write conflicts are allowed, multiple uncommitted versions can coexist under MV3C. Each version involved in a write-write conflict is the output of two possible scenarios. First, there is a predicate P in transaction T that reads the current value of the data object D before updating it. In this case, if another transaction T' with a version for D commits first, T fails validation while matching P against the committed versions of T'. Then, this conflict gets resolved in the Repair phase.

Second, the write is blind and the transaction updates the data object without reading its existing value for the updated fields. In fact, all writes done by a transaction, including the blind ones, are visible to the other transactions only after its commit. Note that the fields of the data object that are used for finding it in the table (i.e., key or indexed fields) do not affect the blind write property, as they remain intact. As was mentioned in section 4.2.3, this assumption holds because the operations that modify the key or indexed fields are not treated as blind write operations and cannot interleave with other blind write operations. Therefore, accepting blind writes does not affect serializability, as the read-set of the transaction remains unmodified at commit time, and the illusion of running the entire transaction at commit time is maintained.

Example 4.2 (continued)**.** Assume in the TransferMoney transaction, there is an extra input parameter named *date* that specifies the date and time of the transaction. Moreover, instead of giving a fee to the central account identified by FEE_ACC_ID, the date and time of the last transaction is updated using the *date* input parameter. That is, substituting the command on Line 17 of Figure 4.2 with the following SQL command:

```
1 UPDATE Account SET t_date=:date WHERE id=:FEE_ACC_ID;
```

This SQL statement translates into a blind write, as the data object is looked up using its primary key *id* and its value field *t_date* is updated without reading its current value. If the central fee account exists and execution is successful, this operation cannot fail in validation unless its predicate P_1 fails. △

Moreover, to keep the process to find the visible value simple, MV3C moves a committed version next to the other committed versions in the version chain. This technique preserves the semantics that uncommitted versions are in the beginning of the version chain, and

committed ones are ordered by their commit timestamps at the end of the version chain. For example, in Figure 4.5, if T_z is committed before T_y, its versions are moved next to the versions created for T_3.

4.2.5 MV3C Repair

If a transaction fails validation, it enters the repair phase. All the read operations in a transaction that run under a timestamp ordering algorithm (such as MV3C or OMVCC) return the same result-sets when re-executed, if the start timestamp does not change. Thus, a transaction that reads obsolete data, if re-executed, would read the same data and fail validation again. Therefore, the first step for repairing the transaction is picking a new start timestamp S' for it. The Repair algorithm, shown in Algorithm 2, is applied with the parameters ($G(S)$, S'), where $G(S)$ is the predicate graph resulting from the initial round of execution using start timestamp S. It uses the results of the Validation algorithm to prune the invalid predicates in the predicate graph.

Algorithm 2 Repair

```
1:  Input: G(S) ← predicate graph resulting from executing a
                  transaction program T using start timestamp S
           S' ← validation timestamp, where S < S'
2:  Output: G'(S') ← repaired predicate graph
3:  (L1, L2) ← Validation (G(S),S')
4:  F ← Set of all nodes in L2 with no incoming edges from L2
5:  for all predicate node f in F do
6:      for all predicate node h in descendant nodes of f do
7:          clear the list of versions in h and remove them from the
              undo buffer of the transaction
8:          remove h from G(S)
9:      clear and remove list of versions in f and remove them
          from the undo buffer of the transaction
10: for all predicate node f in F do
11:     call f.execute(C(f), S'), where C(f) is the closure bound to f
12: return G(S) as G'(S')
```

In line 4 of the Repair algorithm, the set of nodes in L_2 with no incoming edges from L_2 are selected into F, where L_2 is the second part of the output of the Validation algorithm. Based on the selection criterion, it is guaranteed that the nodes in F are not descendants of each other. It can be observed in lines 6 to 8 that a given invalid predicate f in F is *pruned*, which is the process of removing all the versions created by f and its descendants, and then removing the descendants from the predicate graph. After pruning the predicate node f, it is sufficient to re-execute $C(f)$, which is done in line 11. Executing the closure re-instantiates all pruned descendant predicates. The order of re-executing the invalid predicates does not matter, as they are independent. If one is a descendant of the other, the former would have been removed from the graph during the pruning process. In the Repair algorithm, the closures of the valid predicates are not re-executed, as there are no changes in their result-sets.

Example 4.2 (continued)**.** After T_z fails validation, a new start timestamp T_7 is assigned to it. Based on the validation results, P_3'' is the only invalid predicate in T_z. P_3'' reads the version written by T_6 with $bal = 9$ and creates $(T_z, bal, 10)$ as its version. △

After applying the Repair algorithm, the transaction enters the validation phase again. The validation phase is the same as the one for the initial execution of the transaction and has a possibility of success or failure.

Example 4.2 (continued)**.** In the above example, T_z succeeds in the validation phase this time, as there is no concurrent transaction committed during its new lifetime (after T_7). △

In the case of a validation failure in this stage, another round of repair is initiated and this cycle continues until validation is successful. It is important that the whole process of validating a transaction, and drawing a commit timestamp or a new start timestamp depending on the result of the validation is done in a short critical section. However, the repair phase is done completely outside the critical section, as if the transaction is running concurrently with other transactions.

The main purpose of Repair algorithm is rebuilding the same predicate graph as if it is aborted and restarted from scratch. Two predicate graphs are equivalent if there is a graph isomorphism between them and corresponding predicate nodes have the same list of versions attached to them. The following lemmas prove the correctness of the Repair algorithm.

Lemma 4.3. *Given a predicate graph $G(S)$ and an arbitrary node X in it, pruning X and re-executing $C(X)$, the closure bound to X, under the same start timestamp S, rebuilds $G(S)$.*

PROOF. Observe that both the execution and re-execution of $C(X)$ view the same snapshot of the database, as the start timestamp of the transaction is not changed. Since all the versions created by X and its descendants are pruned, any changes created by the closures bound to X and its descendants are removed. In addition, C(X), based on Definition 4.5, is a deterministic function, which guarantees that given the same input, it generates the same output. Therefore, re-executing $C(X)$ re-creates the same predicate graph. □

Lemma 4.4. *Let $G(S)$ be the predicate graph resulting from the execution of a transaction T with start timestamp S that failed validation at timestamp S'. Assume that $G'(S')$ is the predicate graph resulting from applying the Repair algorithm on $(G(S), S')$. Instead of applying the Repair algorithm, if T was aborted and restarted with start timestamp S', resulting in a predicate graph $G''(S')$, then $G''(S')$ is equivalent to $G'(S')$.*

Lemma 4.4 is proven in section 4.7.

4.2.6 Serializability

We claim that the schedules created using MV3C are commit-order serializable. Consequently, it is guaranteed not only that the schedules created by MV3C are serializable, but also that one equivalent serial schedule can be proposed by creating a serial sequence of transactions using

their commit order under MV3C. The serializability of MV3C is stated in Theorem 4.1 and it is proved in section 4.7.

Theorem 4.1. *The committed projection of any multi-version schedule H produced under MV3C is conflict equivalent to a serial single-version schedule H' where the order of transactions in H' is the same as the order of successful commit operations in H and uncommitted transactions are ignored.*

4.3 Interoperability with MVCC

Interoperability is an important aspect of any new concurrency control technique. There are many database and transaction processing systems that are already operating using an existing concurrency control algorithm. Therefore, it is highly desirable that a new concurrency control technique is interoperable with its predecessors, as it facilitates incrementally incorporating the new algorithm into the system by different transactions. This decreases the cost and risk of employing the new algorithm.

The only interaction of an MV3C transaction with other transactions is during the validation phase. The algorithm needs to know about the versions committed during the lifetime of the current transaction, so as to match the predicates of the current transaction against those versions. This is the only piece of information that is required from a previously executed transaction. Consequently, any MVCC algorithm, such as OMVCC, that provides information about the committed versions can seamlessly inter-operate with MV3C. This interoperability provides backward compatibility for free, and makes it possible for the transaction developers to program their new transactions in MV3C, and gradually convert the old transactions in the system into MV3C.

4.4 Optimizations

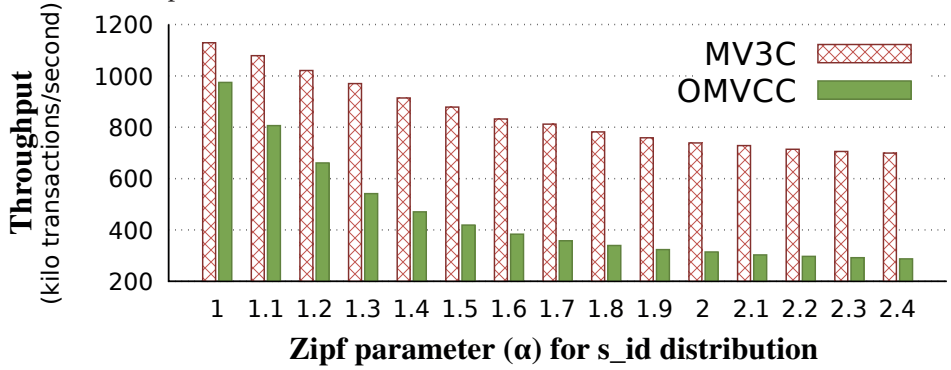
Like MVCC, there can be different flavors of MV3C, each with its own optimizations. This section is dedicated to optimizations that can be employed by MV3C.

4.4.1 Attribute-Level Predicate Validation

To validate an MV3C transaction, the committed versions of all concurrent transactions are considered. These versions are matched (as the whole record) against each predicate in the predicate graph. If a version satisfies a predicate, a conflict is declared. However, it is a pessimistic approach, in that the modified columns in the concurrently committed versions might not be used in the current transaction.

As an optimization, the validation can be done at the attribute-level instead, as in previous works [84]. To enable attribute-level validation, the columns that are used from the result-sets of the predicates are marked for runtime monitoring. In addition, all columns in the data selection criterion of the predicate are monitored. At runtime, each created version stores a list of columns modified in it. Then, in the validation phase, while checking a predicate against a version, the intersection between the monitored columns in the predicate and the

(a) Impact of the number of concurrent transactions (with $\alpha = 1.4$)



(b) Impact of the percentage of conflicting transactions (with 10 threads)

Figure 4.6 – Trading benchmark experiments

modified columns in the version is first computed. If the intersection is empty, there cannot be a match. Otherwise, the predicate-specific match operation is performed.

4.4.2 Reusing Previously Read Versions

The predicates that failed validation are the starting points of the repair phase. After acquiring a new start timestamp in this phase, the failed predicates are re-evaluated, and their result-sets are fed into their assigned closures. Evaluating these predicates from scratch is one approach and it is taken by default in MV3C. However, it is also possible to re-use some of the computation done in the initial execution round that failed validation.

The realization of this optimization depends on the type of predicates and the way they access their target data. However, there are some similarities among all of them. Each predicate that wants to re-use the computation keeps a reference to its result-set. The result-set has to be computed nonetheless, as it needs to be fed into the closure of the predicate. This additional reference is kept only until a successful commit, which is also the end of the lifetime of the predicate. Then, in case of a failed validation, the result-sets have to be fixed for the failed

predicates, a procedure that is predicate-specific. This procedure is merged with the validation phase, as both require matching predicates against versions.

Example 4.4. Consider a predicate that selects data based on a condition over non-indexed columns of a large table. The initial execution is costly, as a full scan over the table is required. However, if this predicate fails, it must be due to a concurrent transaction committing a version that should be a part of the result-set of the predicate. Therefore, the result-set can be fixed by accommodating the concurrently committed versions into the result-set. △

This optimization comes with the cost of keeping a reference to the result-set of each predicate, as well as the procedure for fixing them. Therefore, it is not used by default, and can be enabled per predicate instance. The decision to use this optimization is taken either by the transaction programmer, or an automated analyzer that monitors the failure rate of each predicate. One heuristic is to activate this optimization only for predicates that have a higher failure probability, where fixing the result-set is cheaper than re-evaluation.

4.4.3 Exclusive Repair

In principle, the repair phase of MV3C (cf. section 4.2.5) happens concurrently with other transactions. This design decision can increase the parallelism degree of MV3C, as there is no guarantee about the time required for the repair phase. However, this strategy requires another round of validation after the repair phase is done. As an optimization, it is possible to apply the repair phase in a critical section that prevents other transactions from committing, even though it does not prevent them from running or validating. Thus, an extra validation round can be avoided and the transaction is guaranteed to commit after the repair phase. This optimization should be applied with the utmost care, as it can become a bottleneck in the system. A heuristic is to apply this optimization after N rounds of validation failures after applying the repair phase if the previous repair phases were short.

4.5 Implementation

This section describes the details of our implementation.

Transaction Management. MV3C has a *transaction manager* that is responsible for starting and committing the transactions, similar to the one in [84]. It primarily stores four data items which are shared among all transactions, namely *recently committed* transactions, *active* transactions, *start-and-commit timestamp sequence generator*, and *transaction identifier sequence generator*.

The *active transactions* contains the list of ongoing transactions that have not committed. This list is updated whenever a transaction starts or commits. It is mainly used for tracking the active transaction with the oldest start timestamp, which is needed for garbage collection. Like [84], garbage collection of the versions created by a committed transaction is performed after ensuring that there is no older active transaction that could read the versions. The *start-and-commit timestamp sequence generator* is used for issuing start and commit timestamps. Since both timestamps are issued from the same sequence, to find the transactions that ran

concurrently with a given transaction T, it is sufficient to choose transactions from the *recently committed* list for which the commit timestamp is greater than the start timestamp of T.

The *transaction identifier sequence generator* assigns a unique identifier to each transaction. This unique identifier plays the role of a temporary commit timestamp for an active transaction. The sequence generator starts from a very large number, larger than the commit timestamp of any transaction that can appear in the lifetime of the system. This timestamp is used in the uncommitted versions written by an active transaction. This approach for timestamp assignment simplifies the distinction between committed and uncommitted versions, because the uncommitted versions are not read by any other active transaction.

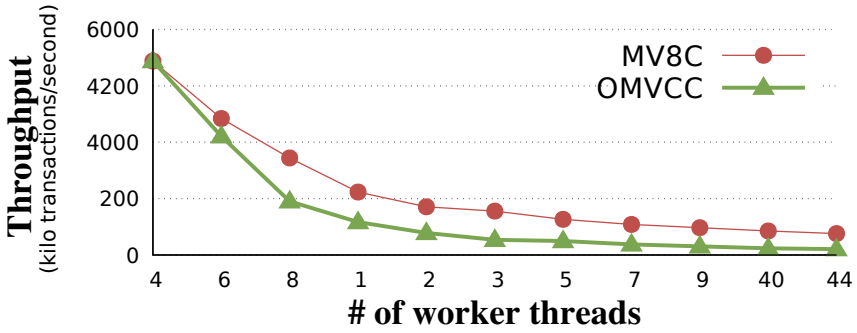
Starting a transaction is done by drawing a start timestamp and a transaction identifier from the corresponding sequence generators. Then, a transaction object is created with these values. The transaction object is an encapsulation of the data needed for running, validating and committing a transaction. Besides the start timestamp and the transaction ID, the transaction object keeps track of its undo buffer and predicate graph. The predicate graph is implemented as a list of root predicates with each predicate storing a list of its child predicates. All operations that interact with the database for reading or writing data require the transaction object, which is passed as a parameter to these operations.

Version Chain Manipulation. All data manipulation operations result in one or more versions. The references to these versions are added to the version chain of the manipulated data object, as well as the undo buffer of the transaction. Each table is implemented as a concurrent cuckoo hash-map [74] of primary keys to data objects. Each data object has an atomic pointer to its version chain called *head*. If multiple uncommitted versions are not allowed for a table (cf. section 4.2.3), the version chain is a simple concurrent lock-free singly linked-list where new versions are added to its *head* or removed from it (in the case of a rollback).

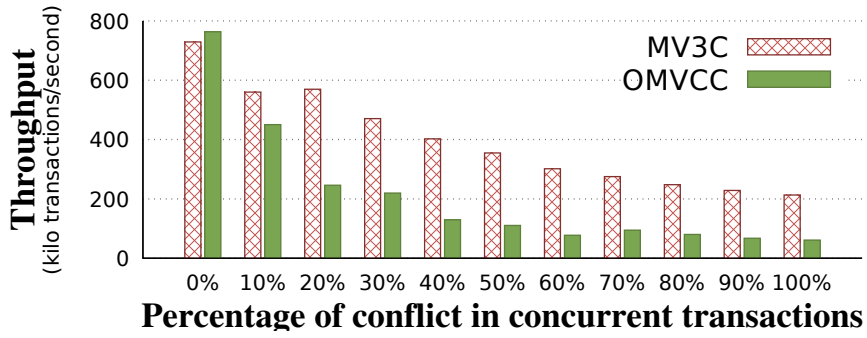
Otherwise, if multiple uncommitted versions are allowed, there are more operations required on the version chain. In this case, a variant of the concurrent Harris linked-list [52] is employed. Each version has an atomic *nextVersion* field. The most significant bit of *nextVersion* is used as *isDeleted* flag. The new versions are always added to the head of the list. If a transaction is rolled back, all *isDeleted* flags of its versions are marked. Otherwise, in a short commit critical section, all its versions are moved next to the previously committed versions in case several uncommitted versions exist (cf. section 4.2.4). The *move* operation is done by marking the version as deleted and creating a duplicate of it and adding it beside the previously committed version if one already exists. The version chain traversal operations (e.g., finding the visible version for a given transaction) are wait-free and responsible for garbage collection of the logically deleted nodes.

Parallel Validation. The list of *recently committed (RC)* transactions is implemented as a concurrent singly linked list. Each committed transaction is added to the head of this list. In the validation phase of a transaction T, the RC list is traversed from its head until the last transaction that committed after T started. T is validated against each of these transactions. Next, T obtains a new timestamp (that would be either a commit timestamp in case of success or a new start timestamp otherwise). Then, if $RC.head$ is not modified (i.e., no new transaction is committed), validation is done. Otherwise, the same procedure is repeated from the new $RC.head$ to the old one.

4.6 Evaluation



(a) Impact of the number of concurrent transactions with 100% conflict ratio

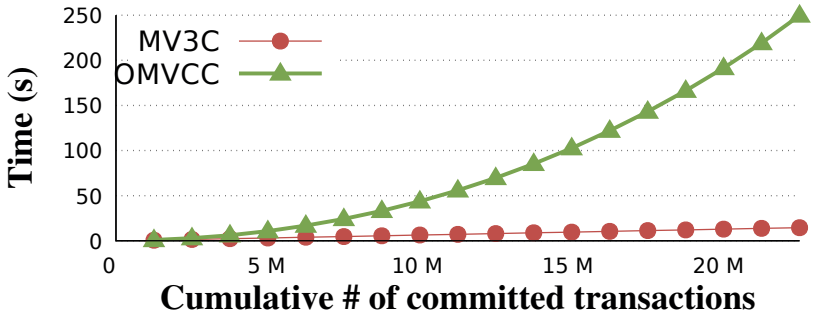


(b) Impact of the conflict ratio with 10 concurrent worker threads

Figure 4.7 – Banking example experiments

We use the TPC-C [118] and TATP [130] benchmarks, as well as some of our own microbenchmarks to show the main pros and cons of MV3C compared to other concurrency control algorithms. We have implemented both MV3C and OMVCC in C++11. The implementation, excluding the tests and benchmarks, for MV3C is 7 kLOC and that for OMVCC is 5 kLOC. Both implementations use row-based storage and redo logs are stored in memory.

Both MV3C and OMVCC are implemented for single and multi-threaded execution. The results of the multi-threaded experiments are shown in this section. The additional experimental results for the single-threaded execution are shown in section 4.9. All experiments are performed on a dual-socket Intel® Xeon® CPU E5-2680 2.50GHz (12 physical cores) with 30 MB cache, 256 GB RAM, Ubuntu 16.04.3 LTS, and GCC 4.8.4. Hyper-threading, turbo-boost, and frequency scaling are disabled for more stable results. We run all the benchmarks five times on a simple in-memory database engine and report the average of the last three measurements. In the multi-threaded experiments, each thread is pinned to a physical hardware core. In all experiments, the variance of the results was found to be less than 5% and is omitted from the graphs.



(c) The ripple effect

Figure 4.7 – Banking example experiments (cont.)

4.6.1 Rollback vs. Repair

The main advantage of MV3C over OMVCC is repairing a conflicting operation instead of aborting and restarting the transaction. There are different parameters that impact the effectiveness of both MV3C and OMVCC, such as the number and size of concurrent transactions, the percentage of conflicts among concurrent transactions, and the average number of times that a transaction is aborted until it commits. We focus on showing the impact of these parameters on the effectiveness of both MV3C and OMVCC. For this purpose, the TATP and TPC-C benchmarks along with two other mini-benchmarks are used. The first mini-benchmark is based on our Banking example (Example 4.2). The second one, named *Trading*, is described next.

Example 4.5. Trading benchmark. This benchmark is the simplified version of the TPC-E [119] benchmark. It simulates a simple trading system, and consists of four tables:

- Security(s_id, symbol, s_price): the table of securities available for trading along with their prices. This table has 100,000 records.
- Customer(c_id, cipher_key): the table of customers along with their encryption/decryption key, used for secure personal data transfer. This table has 100,000 records.
- Trade(t_id, t_encrypted_data): the table for storing the list of trades done in the system. The trade details are stored in encrypted form using the customer's cipher_key, and consists of a timestamp of the trade. This table is initially empty.
- TradeLine(t_id, tl_id, tl_encrypted_data): the table of items ordered in the trades. The details of each record in the last column are encrypted using the customer's cipher_key and consist of the identifier of an asset, and the traded price. The traded price is negative for a buy order. This table is initially empty.

Moreover, there are two transaction programs in this benchmark: TradeOrder and PriceUpdate. A TradeOrder transaction accepts a customer ID (c_id) and an encrypted payload containing the order details. First, the customer's cipher_key is read. Next, using the customer's cipher_key, the payload is decrypted, which contains a sequence number for the t_id, timestamp of the order, and list of securities along with buy or sell flags for each security. Then, the

corresponding securities are read from the Security table to get their current prices and then, the respective rows are added to the Trade and TradeLine tables. A PriceUpdate transaction updates the price of a given security from the Security table.

The instances of these two transaction programs conflict, if a security is requested in a Trade-Order, while concurrently a PriceUpdate is updating its price. In order to simulate the different popularities among securities, the sec_id input parameters in both transaction programs are generated following Zipf distribution. △

Number of concurrent transactions

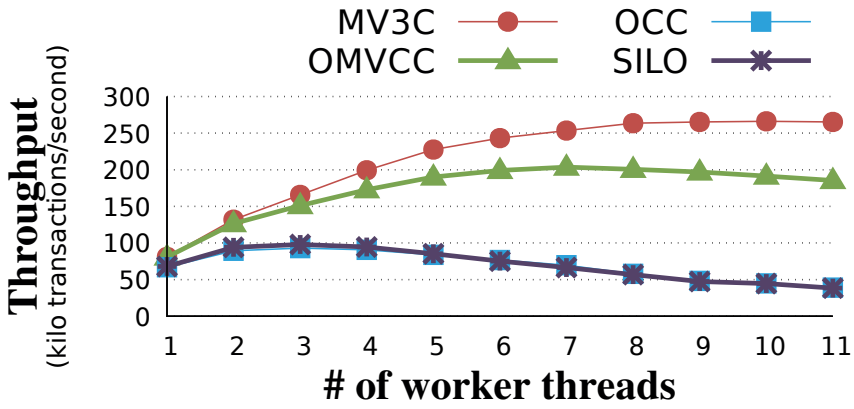
Given a fixed percentage of conflicting transactions in a stream of transactions, if we change the number of concurrent transactions, more conflicts occur. The number of concurrent transactions can be controlled by having a fixed number of worker threads for handling a queue of transactions.

Figure 4.6(a) shows the results of the Trading benchmark (c.f. Example 4.5) with different numbers of worker threads. In this experiment, the distribution parameter α for generating sec_ids is 1.4 for both TradeOrder and PriceUpdate transactions. As shown in this figure, MV3C achieves higher throughput compared to OMVCC, as the contention-level increases. MV3C avoids decryption and deserialization of the input data, and re-executes only a small portion of the conflicting TradeOrder transactions, while OMVCC re-executes the conflicting transactions from scratch. Additionally, PriceUpdate consists of a blind write operation, which does not lead to a conflict in MV3C, but creates a conflict in OMVCC.

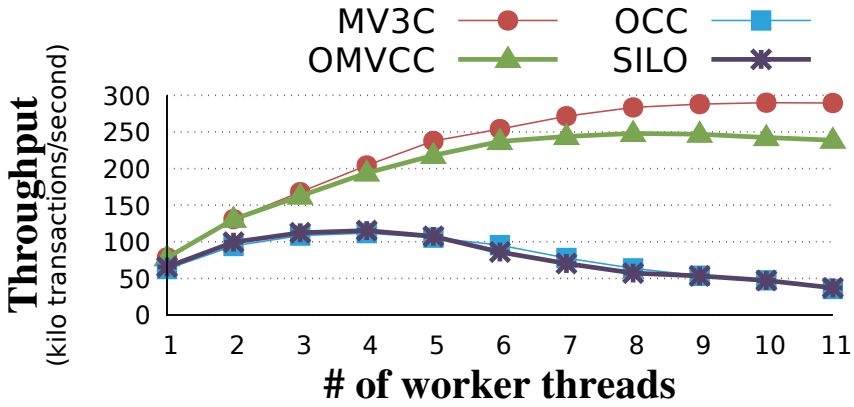
Figure 4.7(a) shows the results of a similar experiment for the Banking example (Example 4.2). In this experiment, there are only TransferMoney transactions, all of which conflict at the end while updating the fee account (line 17 in Figure 4.2). The figure shows the effectiveness of MV3C compared to OMVCC as the time gap for executing 5 million transactions gets wider for higher contention-levels. This experiment illustrates that even though TransferMoney is a small program, the effect of full re-execution in the case of high contention conflicts is not negligible.

TPC-C benchmark. We implemented the TPC-C benchmark using both MV3C and OMVCC with attribute-level predicate validation. Moreover, we used THEDB [131, 132] for the implementation of TPC-C using OCC [69] and SILO [120] algorithms. Almost all conflicting transactions in TPC-C lead to premature abort during execution, instead of reaching the validation phase and failing it. The highest number of conflicts happens in TPC-C when running a smaller number of warehouses. Figures 4.8(a) and 4.8(b) illustrate the performance of MV3C compared to other algorithms for one and two warehouses respectively. In Figure 4.8(a), MV3C shows 30% higher throughput compared to OMVCC with 12 threads. The trend of Figure 4.8(a) continues with more hardware threads. In our benchmark machine, we were limited to 12 physical cores, but running the same experiment in the single-threaded implementation with simulated concurrency shows a 2x performance speed-up for 64 concurrent transactions (cf. section 4.9.2). However, the performance speed-up growth decreases by decreasing the contention, which is shown in Figure 4.8(b). Moreover, the difference between the performance of

MV3C and both OCC and SILO stems from the fact that multi-version concurrency control is a better candidate for the TPC-C benchmark.



(a) Impact of the number of concurrent transactions in the TPC-C benchmark for warehouse=1



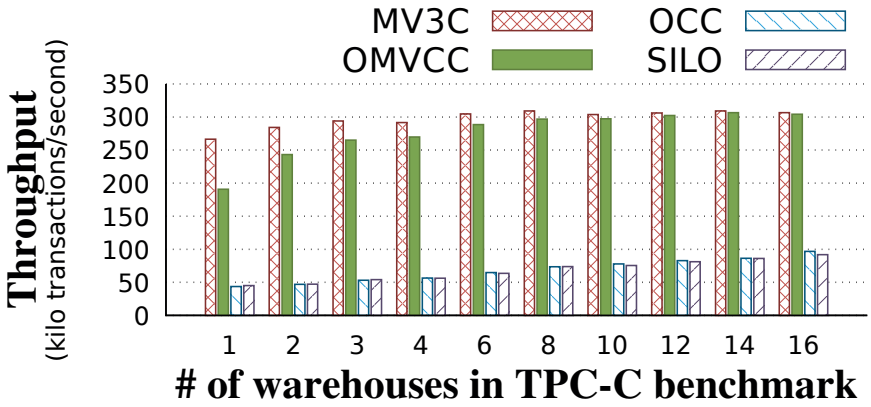
(b) Impact of the number of concurrent transactions in the TPC-C benchmark for warehouse=2

Figure 4.8 – TPC-C experiments

Percentage of conflicts among transactions

For a fixed number of concurrent transactions, the percentage of conflicting transactions determines the success rate between concurrent transactions. If this percentage is 0%, then there are no conflicts. On the other hand, if it is 100%, only one transaction succeeds among all concurrently running transactions; the rest fail, and are repaired or rolled back and restarted depending on the concurrency control algorithm. Consequently, it is expected that MV3C is more effective compared to OMVCC when the percentage of conflicting transactions is higher.

One experiment to illustrate the impact of the percentage of conflicting transactions uses the Banking example. Another program, called NoFeeTransferMoney, is introduced which



(c) Impact of the conflict ratio in the TPC-C benchmark with 10 concurrent worker threads

Figure 4.8 – TPC-C experiments (cont.)

is similar to TransferMoney, but without the fee payment to the central account. In this experiment, we used 10 worker threads and the percentage of NoFeeTransferMoney programs in the mix is varied. Figure 4.7(b) shows that MV3C is more effective than OMVCC as the percentage of conflicting transactions increases.

As another experiment, different α parameters of the Zipf distribution of s_id input parameters in Trading benchmark are used with 10 worker threads. The α parameter determines the percentage of conflicting transactions. The results are illustrated in Figure 4.6(b). This figure demonstrates, once again, that MV3C performs significantly better than OMVCC, as the percentage of conflicting transactions is increased by a larger α parameter.

In addition, as shown in Figure 4.8, by increasing the contention in the TPC-C benchmark (i.e., lower number of warehouses with a fixed number of worker threads) using MV3C becomes more effective compared to OMVCC, OCC and SILO.

4.6.2 Overhead of MV3C

Time overhead. Using MV3C as a generic MVCC algorithm is beneficial, only if it has low overhead in the absence of validation failures or premature aborts. The conflict-free execution is highly probable in low contention scenarios. We consider two approaches for observing this behavior. The first approach executes transactions serially, and the second one runs transactions concurrently with no conflicts among them.

As illustrated in Figures 4.7(a) and 4.6(a), the overhead of MV3C compared to OMVCC in the serial execution scenario (i.e., with a single worker thread) is under 1%. Figure 4.7(b) shows that the concurrent execution of transactions with no conflicts has less than 1% overhead using MV3C compared to OMVCC. This small overhead is mostly related to creating the predicate graph in MV3C, instead of creating a list of predicates as in OMVCC. Applying

compiler optimizations for efficiently compiling the closures bound to predicates in MV3C is necessary to achieve such a low overhead.

Furthermore, in the case where the program has a single root predicate and all conflicts occur in this root predicate during the validation phase, the repair phase involves re-executing the whole transaction. Even in this case, the repair phase of MV3C performs similar to a full rollback and restart (as in OMVCC) without additional overhead as shown by our benchmarks.

Memory overhead. In terms of data structures, compared to OMVCC, MV3C has some additional fields in versions and predicates. For each version, MV3C requires an additional pointer field (8 bytes in a x64 system) for maintaining the list of versions produced inside the closure of its parent predicate. As mentioned in section 4.2.4, this list is used for rolling back the predicates failed in the validation phase. In TPC-C benchmark, the overhead of this extra pointer can be as low as 2% for big records (e.g., Stock table) up to 14% for small records (e.g., History table). The overall overhead of MV3C in TPC-C is 4% extra memory usage.

Moreover, MV3C requires keeping more information for each predicate. The extra fields in a predicate P are used for maintaining the DAG of predicates, the closure assigned to P and the head of the list of versions produced by the closure assigned to P. Accordingly, an MV3C predicate might need twice the memory required for an OMVCC predicate. However, a limited number of predicates are used during the execution of a program, and their memory is reused after the program finishes its execution. For this reason, the memory overhead of predicates is negligible.

4.7 Serializability Proof

Before proving the serializability of MV3C, Lemma 4.4 is restated and proven.

Lemma 4.4. *Let $G(S)$ be the predicate graph resulting from the execution of a transaction T with start timestamp S that failed validation at timestamp S'. Assume that $G'(S')$ is the predicate graph resulting from applying the Repair algorithm on $(G(S), S')$. Instead of applying the Repair algorithm, if T was aborted and restarted with start timestamp S', resulting in a predicate graph $G''(S')$, then $G''(S')$ is equivalent to $G'(S')$.*

PROOF OF LEMMA 4.4. Assume that the database is duplicated at timestamp S' into two identical instances, D_1 and D_2. On instance D_1, T aborts and restarts using the new start timestamp S', resulting in the predicate graph $G''(S')$. On instance D_2, the Repair algorithm is applied on $(G(S), S')$, which results in the predicate graph $G'(S')$.

$G'(S')$ and $G''(S')$ have the same root predicates, without considering their lists of versions. The reason is that the creation of a predicate depends on its parent predicate. The root predicates do not have a parent by definition, so they are created regardless of the database state. It is also guaranteed that they are not removed from the predicate graph by the Repair algorithm, as they are not descendants of any other nodes. Moreover, in line 3 of the Repair algorithm, the results of the Validation algorithm on $(G(S), S')$ are used (i.e., L_1 and L_2). Let X be an arbitrary node selected from $G(S)$. There are three cases.

Case 1: X is in L_1. Then, X is a valid predicate, and based on Lemma 4.1, all of its ancestors are valid, too. Let Y be the list of root predicates that are the ancestors of X. Nodes in Y are common in both $G'(S')$ and $G''(S')$. Hence, as part of executing T on D_1, the corresponding nodes in $G''(S')$ are created and the closures bound to them are executed. As nodes in Y are valid nodes, they read the same data as S, and take the same program flow, which results in creating X with the same list of versions inside it.

Case 2: X is in L_2 and has no incoming edge from another node in L_2. Then, either X has no parent, or its parents are in L_1. In both cases, X is created while executing T from scratch on D_1. If X has no parent, then it exists regardless of executing under timestamp S or S'. If X has a list of parents Y in L_1, then based on Case 1, all nodes Y exist in $G''(S')$ and they are valid. As the execution of their closures using the timestamp S' is the same as S, they create the same child predicates and X is among them. Lines 5-11 of the Repair algorithm prune X and re-execute it. Then, the closure bound to X is re-executed in line 11 of the Repair algorithm using timestamp S', which is identical to executing it on D_1. And, it results in the same descendant nodes and list of versions for X in both $G'(S')$ and $G''(S')$.

Case 3: X is in L_2 and has an incoming edge from another node in L_2. In this case, X has an ancestor, Z, from L_2 that falls into Case 2. Thus, Z is pruned and X is removed from $G(S)$ in lines 7 and 9 of the Repair algorithm. As it is shown in Case 2, the same descendant nodes are recreated for Z in both $G'(S')$ and $G''(S')$.

Thus, $G'(S')$ and $G''(S')$ are equivalent. □

Next, the serializability of MV3C that is claimed in Theorem 4.1 in section 4.2.6 is proven after restating the theorem.

Theorem 4.1. *The committed projection of any multi-version schedule H produced under MV3C is conflict equivalent to a serial single-version schedule H' where the order of transactions in H' is the same as the order of successful commit operations in H and uncommitted transactions are ignored.*

PROOF OF THEOREM 4.1. Based on Definition 4.3, the versions created by a transaction T become visible to other transactions only after T commits. Accordingly, the operations done by uncommitted transactions cannot affect the serializability of committed transactions. Thus, the uncommitted transactions can safely be ignored, and only committed transactions are considered in this proof.

By showing that all dependencies between different transactions executed under MV3C have the same order as their commit timestamp, it can be proven that any execution of transactions under MV3C is serializable in commit order. Read-only transactions read all committed versions that are committed prior to their start. Moreover, the commit timestamp of a read-only transaction is the same as its start timestamp, as if it executed at that point in time.

An update transaction starts, executes, and then enters the validation and commit phase. If the validation is successful, a commit timestamp C is assigned to the transaction. Otherwise, the transaction acquires a new start timestamp S' and enters the repair phase. By Lemma 4.4, the predicate graph resulting from applying the Repair algorithm at S' is the same as the one

resulting from aborting and restarting the transaction from scratch at timestamp S'. Then, the repaired transaction enters the validation phase again, and this cycle continues until the transaction succeeds in the validation phase.

Thus, it is sufficient to prove that for an update transaction that starts with timestamp S, executes, successfully passes the validation phase and commits, the visible effect of the transaction is as if it is executed at one point in time, C, where $S < C$. The proof is done by contradiction, and is similar to the serializability proof in [84]. There are some modifications for allowing write-write conflicts in MV3C, as this kind of conflict leads to premature abort and restart in [84].

Let T_1 be an update transaction from the committed projection of H with start timestamp S_1 and commit timestamp C_1. Suppose that the execution of T_1 cannot be delayed until C_1. Then, there is an operation, $o1$ in T_1 that conflicts with an operation $o2$ in another transaction, T_2 (started at S_2 and committed at C_2), with $o1 < o2$ and T_2 committed during the lifetime of T_1, i.e., $C_2 < C_1$. There are four cases corresponding to the possible combinations of two operations $o1$ and $o2$.

Case 1: both are reads. This case contradicts the assumption that $o1$ and $o2$ are conflicting.

Case 2: $o1$ is write and $o2$ is read. Based on Definition 4.3, the version written by $o1$ is not visible to $o2$, as $S_2 < C_1$. Thus, it contradicts the assumption that $o1$ and $o2$ are conflicting.

Case 3: $o1$ is read and $o2$ is write. By C_1, the version V_2 written by $o2$ is committed and exists in the undo buffer of T_2, as $C_2 < C_1$. The operation $o1$ is done via a predicate P_1. Hence, while validating T_1 at C_1, the Validation algorithm matches P_1 against V_2. If P_1 matches, T_1 fails validation and cannot commit at C_1. This contradicts the fact that T_1 with this commit timestamp is in the committed projection of H. Otherwise, if P_1 does not match V_2, then it contradicts the assumption that $o1$ and $o2$ are conflicting.

Case 4: both are writes. In this case, $o1$ and $o2$ can be swapped and this contradicts the assumption that $o1$ and $o2$ are conflicting. Basically, an individual write operation acts as a blind-write. Thus, the blind-write operation in $o1$ can be delayed until C_1. If a write operation is not a blind-write, there is a read operation done before it, and this case is converted to either Case 2 or Case 3. □

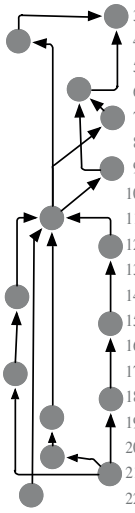
4.8 Translating into MV3C DSL

To execute a transaction under MV3C, the transaction program can also be given in the form of a PL/SQL-like language. In this case, the program goes through a translation pipeline in order to be converted into the MV3C DSL described in section 4.2.2. There are three stages in the compilation pipeline, described next.

Stage I. In this stage, the representation of the program is converted into a *dependency graph*. The dependency graph, also known as program dependence graph representation [39], is a graph where nodes are program operations, and an edge from node A to node B means the operation in node A depends on the results of the operation in node B in order to execute.

The operations provided in the PL/SQL-like language can be categorized into read, computation, data manipulation, or a combination of these types. As MV3C DSL does not have combined operations, the combined operations are decomposed into primitive ones during the construction of dependency graph. This decomposition is done in such a way that preserves the dependencies. It should be noted that read operations in the graph contain the data selection criteria that is represented by a predicate in MV3C DSL. Consequently, all the read operations in the graph are marked as predicate nodes.

The dependency graph is just another representation for the actual program with more data locality. In the dependency graph, the operations for reading data come closer to the operations for manipulating and consuming the data. In addition, the marked predicate nodes in the graph form an overlay graph built on top of the dependency graph. The *predicate overlay graph* is a DAG where nodes are predicates and there is an edge from node p1 to node p2 if there is a path in the dependency graph from p1 to p2. As stated earlier in section 4.2, the versions created by an uncommitted transaction are hidden from other transactions. Thus, the only reason for a transaction T to fail in the validation phase is reading some outdated data objects, which were committed during the execution of T. Therefore, the predicate overlay graph can be viewed as the possible failure points in the program.



```
1  /* fm = from, acc = account and bal = balance */
2  TransferMoney(fm_acc, to_acc, amount) {
3   START;
4   SELECT bal INTO :fm_bal FROM Account WHERE id=:fm_acc;
5
6   IF(amount < 100)
7       fee = 1.0;
8   ELSE
9       fee = amount * 0.01;
10
11  IF(fm_bal > amount+fee) {
12   SELECT bal INTO :to_bal FROM Account WHERE id=:to_acc;
13
14   fm_bal_final = fm_bal - (amount + fee);
15   to_bal_final = to_bal + amount;
16
17   UPDATE Account SET bal=:fm_bal_final WHERE id=:fm_acc;
18   UPDATE Account SET bal=:to_bal_final WHERE id=:to_acc;
19   SELECT bal INTO :c_b FROM Account WHERE id=:FEE_ACC_ID;
20   UPDATE Account SET bal=:c_b+:fee WHERE id=:FEE_ACC_ID;
21   COMMIT;
22  } ELSE ROLLBACK;
23 }
```

Figure 4.9 – An example transaction program in a PL/SQL-like language with its dependency graph.

Example 4.2 (continued)**.** The result of applying the first stage on the transaction program shown in Figure 4.2 is illustrated in Figure 4.9. The dependency graph of the program is shown on the left-side of the Figure 4.9, where the operation on each line is represented by a gray circle, and arrows between circles show the dependency among the operations. In addition, the combined operation (read and write) on line 17 of Figure 4.2 is converted into two primitive operations (one for read and one for write) on lines 19 and 20 of Figure 4.9. △

Stage II. In the second stage of translation into MV3C DSL, the dependency graph is partitioned into sub-graphs satisfying the following properties:

- Each sub-graph has to have at least one predicate node inside it, as it is the representative of an operation that can fail validation. The predicate nodes form the roots of the sub-graphs.
- For all nodes in the sub-graph, every node that depends on it is also in the sub-graph. This property ensures that the sub-graph represents the minimal set of operations that have to be re-executed if the predicate fails.

A *failure unit* is the smallest sub-graph (with respect to the number of vertices) satisfying the above properties. The translation process aims at partitioning the dependency graph into *failure units* in a bottom-up manner.

The above procedure is repeated recursively after collapsing the *failure units* that are already found into black-box nodes (i.e., the predicates inside these failure units are not considered anymore), until predicates appear only as roots. The output of this process is the nested partitions of the graph, where each partition can fail the validation phase. If there are other predicates depending on the root predicate nodes, they are in a nested sub-graphs.

Example 4.2 (continued)**.** After applying the second stage on the dependency graph shown in Figure 4.9, two failure units are found initially. These two failure units are illustrated in Figure 4.3 using dark gray color, which are represented by predicates P_2 and P_3. Then, these two failure units are collapsed into two black-box nodes; stage II runs recursively and in the next round, another failure unit is found. This failure unit is shown in Figure 4.3 using light gray color, which is represented by predicate P_1. △

Stage III. In the final stage of translation, the transaction program is generated using the dependency graph that is partitioned into *failure units*. The generated program has the following characteristics.

- It is equivalent to the input program, as the program logic should remain intact.
- Corresponding to each *failure unit* in the dependency graph, there is a part of the generated program that contains the responsible predicates. This part of the program depends on the results of those predicates.
- Each data manipulation operation is given the references to its direct parent predicates. These references are required for cleaning up the conflicted state, in the case of a conflict happening in the program represented by this *failure unit*.
- It has to build the predicate dependency graph, when the generated program is executed.

It should be noted that the predicate overlay graph in the previous stage is not necessarily equivalent to the runtime predicate dependency graph, even though those are similar. The difference is that the runtime predicate dependency graph has the actual values for predicate parameters while compile-time predicate dependency graph contains symbols for them. In addition, a predicate (and its assigned closure) can be instantiated several times at runtime, e.g., when it is evaluated in a loop construct. The program generated using each *failure unit* is named a *closure transition*, as it encloses all the operations that depend on one or more predicates, along with the context variables required for executing it.

The algorithm for generating the transaction program using the partitioned dependency graph starts with the top-level *failure units*. By definition, there is no overlap between the *failure units* and therefore, the generated code for them does not overlap either.The code generation involves the following three steps :

1. The root predicates are extracted from a *failure unit*.
2. The code for instantiating the root predicates is generated.
3. The code for the nodes dependent on the root predicates is generated and wrapped inside a closure. This closure is passed to the *execute* function of the root predicate. The dependent nodes are computational nodes, data manipulation nodes or nested *failure units*.
   - Generating code for computational nodes, which does the intended computation, is straightforward.
   - For data manipulation nodes containing insert, update or delete operations, the parent predicates have to be correctly registered in the generated code.
   - For each nested *failure unit*, the same procedure is repeated recursively. The only difference is that, while generating code for predicates in the nested *failure units*, their direct parent predicates have to be indicated. Moreover, the returned results of the parent predicates can also be used inside the operations of the nested *failure units*.

4.9 Extended Evaluation

A common approach to evaluate a concurrency control technique is having a single thread of execution while concurrency among transactions is simulated. The same approach is taken in [84] while evaluating OMVCC. This technique leads to more deterministic results, avoids the overheads of concurrent execution and focuses on showing the impact of the concurrency control algorithm. Moreover, the number of concurrent transactions is not bound to the number of available CPU cores on the machine used for experiments. The concurrency among transactions is realized in these implementations by dividing programs into smaller pieces and interleaving these pieces with ones from other programs. Then, at each step, a single transaction piece gets executed.

A program can be divided into at least three pieces: (1) the start transaction command, (2) the logic of the transaction program itself, and (3) the commit transaction command. In addition, the logic of the transaction program can be further divided into smaller pieces. However, for MV3C transactions, the concurrency level only depends on the number of active transactions (i.e., the transactions that started but are not committed) rather than the number of interleaved program pieces, because transactions only see the committed state. In our experiments, unless otherwise stated, we use the notion of *window* to control the number of concurrent transactions, N. Given a window size N, N transactions are picked from the input stream and all of them start, then they execute, and finally they try to validate and commit one after the other. Here, the transactions that fail during the execution are rolled back and moved to the next window. Transactions that fail validation acquire a new timestamp immediately and their executions are moved to the next window. If $N = 1$, then the execution is serial.

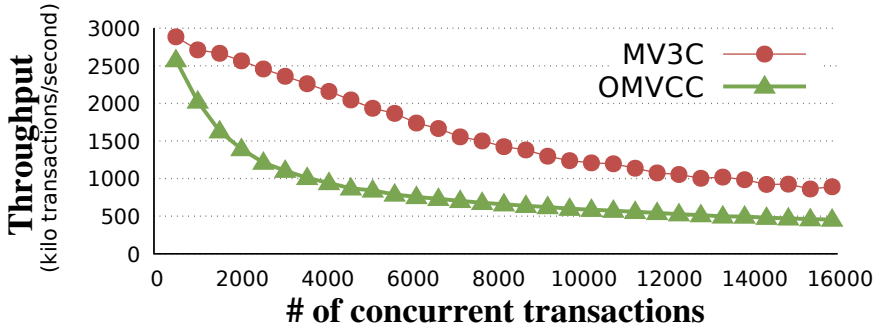


Figure 4.10 – TATP experiment with scale factor 1

4.9.1 TATP Benchmark

We have implemented the TATP benchmark in both MV3C and OMVCC. Figure 4.10 shows the results of executing 10 million TATP transactions with different window sizes. This experiment is done for scale factor 1 with non-uniform key distribution and attribute-level validation. The input data is generated using OLTPBench [33]. Once again, this experiment confirms the increased effectiveness of MV3C compared to OMVCC, as the window size increases. Note that even with non-uniform key distribution, the number of conflicting transactions for small window sizes is low, as 80% of the workload consists of read-only transactions. Thus, MV3C and OMVCC show similar results for small window sizes; the difference can only be seen in bigger windows. Moreover, as transaction programs in the TATP benchmark are very small. The main advantage of MV3C over OMVCC is that the former optionally accepts write-write conflicts, while the latter prematurely aborts after a write-write conflict. This decision leads to having no conflicts among Update_Location transaction instances in MV3C, while it leads to aborts in OMVCC.

4.9.2 TPC-C Benchmark

In addition to the multi-threaded experiments for TPC-C, we conducted further experiments with simulated concurrency among transactions. In the simulated concurrency, we are not limited to the number of physical cores available in the system. Figure 4.11 illustrates the results of running TPC-C up to window size 64. The results shown in Figure 4.11 are interesting, not only because it confirms the effectiveness of MV3C compared to OMVCC as the number of concurrent transactions increases, but also because its relative difference compared to OMVCC matches the multi-threaded results shown in Figure 4.8(a).

4.9.3 The Ripple Effect

The time saved by repairing the transactions instead of aborting and restarting them from scratch can be used to reduce the load on the system. To show this effect, we ran an experiment where two streams issue transactions at a constant rate. The first stream issues its transactions

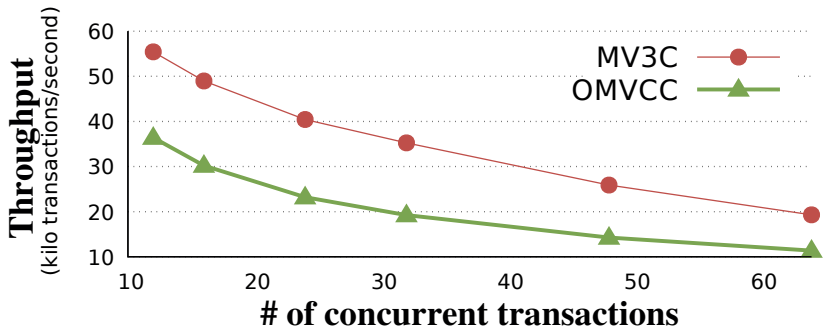


Figure 4.11 – TPC-C experiment with 1 warehouse

at almost the serial transaction processing rate, while the second issues its transactions at a much slower pace.

Figure 4.7(c) shows the results of one such experiment. In this experiment, the TransferMoney transaction program from the Banking example (Example 4.2) is used in both streams. The schedule is generated using logical time units. The time taken for executing one TransferMoney is assumed to be 250 units for both algorithms. We choose three quarters of this time, i.e., 187 units as the time for partially re-executing conflicting blocks in MV3C, based on the results of Figure 4.7(a). The two streams produce TransferMoney transactions at the rate of 251 units and 72,000,000 units respectively. This figure shows not only that the time for processing MV3C transactions is lower, but also that the overall behavior of MV3C is completely different than OMVCC. Basically, this experiment shows that a longer conflict resolution approach not only affects an individual transaction, but also has a compound effect on subsequent transactions. The reason is an increase in the probability of having many concurrent transactions, which results in even more conflicts.

5 Related Work

Program analysis and compilation techniques in the programming languages community and the transaction processing in the database community are both fundamental areas of research. Work in these area has resulted in many publications and books [7, 44, 127]. This section covers work done on topics related to both Beta and MV3C.

5.1 Front-end DSLs

The integration of queries into a programming language, initiated by [77], has been adopted by many languages like Scala (Slick) and Ruby. Recent advances like SWITCH [45, 76], Ferry [47] and Sloth [24] are compiled in the client and produce optimal SQL (with a minimal number of executed queries). Our DSL in Beta does not generate queries on the client side but compiles transaction programs directly in the database server and outputs low-level code specialized for a given transaction set.

5.2 Database Compilers

Emerging database systems aim at compiling queries (HyPer [83], MemSQL, QPipe [51], NoDB/ViDa [2, 59, 60], DBLab/LegoBase [103, 64]), transactions (DORA [87], ADDICT [117], H-Store/VoltDB [109]) or the whole system (Tupleware [30]). Only Hekaton [32] integrates compilation of transactions from a high-level language (SQL). The main differences of Hekaton with Beta are:

- *Compilation*: Beta uses a unified DSL with two abstraction levels, Hekaton uses two disjoint internal representations, i.e., mixed abstract tree (MAT) and pure imperative tree (PIT). Hekaton inlines and wires iterator functions using `gotos`, which still keeps some intermediate materializations; with fusion and CPS, we get rid of them.
- *Data structures*: Hekaton supports predefined indexes (Bw tree, hash with fixed number of buckets); Beta chooses the most appropriate dynamic data structure based on a restricted set of queries.

- *Environment*: Hekaton fixes the concurrency model to MVCC, but Beta handles multiple concurrency models (see section 3.7). Hekaton compiles directly to C whereas Beta relies on multiple optimization layers and as the last stage generates C code.

HyPer [83] generates low-level code using LLVM to improve query performance. The system is different from Beta in that HyPer only focuses on individual transaction programs and does not even compile whole transaction programs, but individual queries. Furthermore, Beta leverages its modular design to give insight into the performance impact of each individual optimization (as shown in the experiments), while such a facility is not reported for HyPer.

5.3 Application-Level Compilation

Database applications are commonly implemented using a combination of procedural and declarative languages. The business logic is usually implemented in an imperative language, such as C# or Java, the data access layer is programmed using SQL, and an application programming interface (API) such as Java Database Connectivity (JDBC) is used to communicate the SQL queries between the Java program and the database. Having these two environments can degrade performance. The main reason is that neither of these environments can make use of the optimization opportunities in the other side. For example, the database indexes are not known by the Java environment and the database system is not aware of the loops in the Java code [105].

One way to optimize such programs is by using program analysis techniques to extract declarative queries from the imperative code [26, 25, 22, 128, 129]. As a result, the extracted code can benefit from the optimizations offered by the underlying database system. Furthermore, it is possible to partition database applications between the application runtime and the database system [22, 23], merge several related queries into a single query [49, 75], and prefetch the query results [91, 21]. However, as SQL is not as expressive as an imperative language, this approach is not applicable to all database applications. In addition, to apply optimizations available at a lower level of abstraction (e.g. operator inlining, inter-operator optimization, etc.), one should rely on the database system.

The alternative approach is to rewrite both the application logic as well as the data access part into an intermediate language, such as UniQL [105] and forelem [97]. This way, all the optimizations happening in *both* the application runtime (e.g. the underlying optimizing compiler of the application program) and the database system (e.g. query optimization) become applicable directly. Although the intermediate language in such systems is expressive enough, these systems mainly focus on high-level optimizations available at the corresponding intermediate language [105]. In contrast, our approach utilizes multiple levels of abstraction and, thus, makes it possible to perform optimizations available across different abstraction levels, such as `Record Structure Specialization` (cf. section 3.4.6).

There were several efforts to boost the performance of the database applications written using *language integrated queries* (LINQ [77]) using database-inspired strategies and optimizations through code generation and just-in-time compilation [46, 81, 82, 124]. In general, all these techniques employ compilation to convert high-level LINQ programs to more efficient, imperative low-level code. This line of work mostly targets making query processing of collections in

the memory space of the application more efficient by leveraging database technology. As an example, [124] can modify the memory layout of a collection of records, from a generic array of pointers to objects allocated on the managed heap, into an array of contiguous objects. However, due to the lack of multiple intermediate languages in these systems, it is not possible to support `Record Structure Specialization`, `Mutable Records`, etc.

5.4 Multi-Version Concurrency Control

Concurrency control techniques from the MVCC family (which includes snapshot isolation) are de facto standards [32, 89, 106] in open source as well as commercial database management and transaction processing systems. PostgreSQL [89], Microsoft SQL Server's Hekaton [32], SAP HANA [106], HyPer [61], and ERMIA [63] are among these systems. In addition, there is recent work proposing efficient MVCC algorithms for in-memory transaction processing [71, 78, 84, 63]. As was mentioned in section 4.1, MV3C has an efficient conflict resolution mechanism, which is missing in the existing MVCC algorithms.

For the scenarios that are interesting for MV3C (i.e., high contention data objects and long running transactions), one could propose pessimistic concurrency control algorithms that make use of locking. There is a large body of research on the use of locks in different concurrency control algorithms. The one used with multi-versioning is Multi-Version concurrency control with Two-Phase Locking (MV2PL) [6, 18]. However, a pessimistic approach to concurrency control yields low performance for long running transactions, as the acquisition of a highly demanded resource by a long running transaction requires either preventing the long-running transaction from committing, or stopping almost all other transactions.

5.5 Partial Rollback and Restart

The problem of avoiding a complete rollback and restart of a failed transaction has been there since the early days of transaction processing field. Sagas [40] is a pattern for writing long-running transaction programs where transaction programs are divided into smaller actions all of which have a corresponding undo action. Then, actions run sequentially. Each action releases the database resources after it is done. If an action fails, it is enough to roll it back and undo the previous actions until a safe point is reached and then, retry the transaction. This approach is a coarse-grained solution and can be used irrespective of the concurrency control algorithm used inside each action. MV3C is a more fine-grained solution to solve the issue inside one of these actions. Automating the conversion of an arbitrary transaction to Sagas is not straightforward and requires programmer effort, while MV3C provides a mechanism to automatically convert transaction program into MV3C DSL and does not require programmers to provide the undo actions. Another major difference with MV3C is that a Saga is not a transaction, as it does not guarantee isolation between its actions.

Another approach to transaction repair is proposed in [123]. This method requires the transactions to be written in a functional language similar to Datalog. As transaction programs are functional, their impact can be summarized as *delta changes*. Then, in the case of a conflict, it uses the delta changes of committed transactions to fix the conflicted transactions by re-

peatedly employing the incremental leapfrog triejoin [122] until a fixpoint is reached. This approach mainly focuses on repairing heavy joins inside the transactions.

Transaction Healing (TH) [132] is another solution that tries to resolve conflicts among optimistic concurrency control (OCC) transactions at commit time. Despite approaching the same problem (i.e., repairing conflicts between transactions) MV3C and TH differ in several aspects. First, TH works for OCC while MV3C targets MVCC. This choice retains all the differences between OCC and MVCC, one of which is MV3C does not block read-only transactions while TH might block them during the validation, healing and commit phases. Second, the validation phase of TH ignores the semantics and predicates of the program. Similar to OCC, it checks the whole read-set of the program, so a big read-set can increase its validation time. However, MV3C considers the read predicates of the program for validation, regardless of the number of read data objects using that predicate. Third, the healing phase in TH is always done in a critical section under the assumption that healing phase is always short. This is not a valid assumption for a general purpose concurrency control algorithm in which the healing phase might be as expensive as the re-execution of the whole transaction. However, MV3C avoids applying the repair phase in a critical section by default, even though it is still a possible option (cf. section 4.4.3). Fourth, TH is more limited in the type of input transaction programs than MV3C, in that it requires the dependency graph among transaction operations to be statically known. This condition means the approach cannot be used with arbitrary transaction programs with arbitrary control-flow statements inside them. By contrast, even though MV3C uses the available dependency information at transaction compile time, it does not limit the control-flow statements inside transaction programs. MV3C achieves this flexibility by constructing its predicate graph at runtime based on the dependency information encoded in the transaction program, rather than only depending on static dependency information.

5.6 Nested Transactions

In the nested transaction model [4, 79, 38], a transaction consists of primitive actions or sub-transactions, which can again be nested transactions. In this model, after detecting a conflict inside a sub-transaction, only the sub-transaction is aborted, the state at its start time is recreated, and it is re-executed. The boundaries of such sub-transactions are user-defined, and checkpointing is the usual technique used for recreating the state.

Irrespective of the operations done in the sub-transaction, checkpointing has to be pessimistic. A checkpoint can be used anywhere within the program and no assumptions about the program can be made during its creation. In addition, the number of extra computation cycles required for checkpointing is not negligible. It requires capturing the whole execution state, and this work is not beneficial to the execution of the main transaction program. Moreover, all work done in a conflicting sub-transaction is lost, and cannot be reused in the subsequent re-execution of the sub-transaction.

MV3C achieves higher throughput than the generic nested transaction model due to some key differences. MV3C has stricter regulations that do not allow an arbitrary splitting of programs. Instead, the boundaries are defined based on the possible failure points in the program and the dependencies among the operations. Consequently, faster and more compact checkpointing is achieved, as MV3C tailors the checkpoint for each sub-transaction specifically, unlike the

generic checkpointing used in the nested transaction model. Also, MV3C does not commit its so-called sub-transactions, and only tries to commit the transaction as a whole. Thus, MV3C incurs low overhead for executing transactions while still being able to efficiently repair conflicting transactions. This is possible only because of the constraints in the definition of boundaries for the so-called sub-transactions in MV3C.

5.7 Coordination Avoidance

Coordination avoidance (CA) is an approach proposed in [96] and generalized in [5]. In this approach, the semantics of a transaction program is analyzed and the flexibilities in the semantics are used to avoid unnecessarily declaring conflicts between transactions. For example, in a flight reservation system, the number of *remaining seats* in a flight is stored for each flight record. This field is decremented by *flight reservation* procedure calls. Concurrent updates from several *flight reservation* invocations into the *remaining seats* field do not conflict, as long as at least one seat is available and this is the only conflicting operation. This results in schedules that are correct but not necessarily serializable.

In general, even though both MV3C and CA approaches tend to increase transaction processing performance, they have the following differences: First, unlike CA, MV3C produces serializable schedules. Second, CA requires knowing all the transaction programs in advance in order to analyze their semantics. On the contrary, using MV3C does not limit the system to run a set of pre-determined transaction programs, and it can even interoperate with other concurrency control algorithms (cf. section 4.3). Third, as explained in section 4.8, MV3C has a systematic way of analyzing transaction programs. In spite of that, CA approaches are guidelines for optimizing the execution of transactions at user-level and cannot be fully automated.

5.8 Timestamp Allocation

OCC uses timestamps for correctly validating the concurrently executed transactions [69]. The mainstream approach in OCC and its refined variants [69, 95, 14] is to allocate a start timestamp when a transaction starts and a commit timestamp at commit time. These timestamps are assigned by a centralized timestamp allocator. In the multi-version variants of it, both MV3C and OMVCC do a similar job, as explained in section 4.2.1.

However, as it is shown in [134], a central timestamp allocator can become a bottleneck and limit the scalability of the timestamp-based algorithms. There are several approaches to overcome this bottleneck. Silo [120] tries to solve this issue by allocating coarse-grained (every 40ms) global timestamps. Then, within a timestamp, the transactions are ordered based on their read-after-write dependencies. Another approach taken by TicToc [135] is calculating the commit timestamp of a transaction only based on the timestamp of the data objects read or written by it. This approach not only removes the central timestamp allocation issue, but also creates more opportunities for transactions to commit compared to other OCC approaches. However, unlike MV3C, the resulting schedule of TicToc is not commit order serializable [135]. Moreover, TicToc suffers from a phantom anomaly if it wants to perform efficiently [135], while MV3C does not have this issue because of its predicate-based validation mechanism.

Even though both Silo and TicToc try to increase the transaction processing throughput by decreasing unnecessary contention, they still do not have any solution for real contention among transactions. If a transaction fails to validate, they simply abort and restart it from scratch. This not only incurs a performance penalty, but also is unfair against read-mostly transactions [63]. The main issue that MV3C tries to solve is having a low-overhead repair mechanism that is missing from both Silo and TicToc. We believe that the combination of these approaches can lead to new concurrency control techniques that have the best of both worlds.

6 Conclusion and Future Work

The motivation behind this research is the ever increasing need for having more efficient data processing pipelines to support existing applications and allow new applications in the future. In this thesis, we took the approach of software specialization and proposed a compilation suite for database applications and transaction programs, named Beta.

Beta uses a domain-specific language (DSL) to write transaction programs; it decomposes into functional and imperative parts that relate respectively to queries and query plans, yet share a single intermediate representation. In Beta, the general compilation techniques (deforestation, CPS, CSE, etc.) and domain-specific optimizations are combined in a staged compiler to produce efficient code. Besides, we propose an efficient runtime engine to be the backend of the programs generated by Beta. We show that a holistic compilation approach, such as Beta, allows optimizations that are not possible otherwise.

We applied extensive experiments on the ideas presented in this thesis to demonstrate that in the case of the TPC-C benchmark, Beta produces code competitive with highly optimized hand-written implementations of transaction programs. Also, we experimented with compiling the stored procedures for incremental view maintenance. The code generated by our compiler outperforms the DBToaster trigger compiler by one order of magnitude on TCP-H.

Moreover, in this thesis, we explored the possibility of using program analysis and compilation techniques inside the concurrency control algorithms. We proposed MV3C, a transaction repair and conflict resolution mechanism for optimistic MVCC, proved its correctness and showed its effectiveness in various scenarios. Depending on the situation, MV3C can achieve more than an order of magnitude improvement in transaction processing throughput, while in the worst-case does not have any significant overhead compared to optimistic MVCC.

Future Work. There is a multitude of opportunities and future research directions in the area of using program analysis and compilation techniques for improving the performance and usability of database systems. We briefly discuss some of these directions next.

First, we briefly addressed the process of handling application updates in presence of an optimizing compiler like Beta in section 3.2. However, more research is needed to find the best approaches to handle application updates, especially the ones that involve changes to the database schema or the type of workload.

Second, the dependency information captured by MV3C DSL (see section 4.2) can also be leveraged to achieve other performance improvement goals. As an example, logically parallel branches of execution (similar to Figure 4.1(a)) are easily identified in a program encoded in MV3C DSL. Then, a transaction execution engine can benefit from this dependency information to introduce intra-transaction parallelism, i.e., parallelizing the execution of a single transaction to achieve a lower latency.

Third, the idea of transaction repair can also be extended to pessimistic and lock-based concurrency control algorithms. For example, in the case of detecting a deadlock in a locking algorithm, instead of aborting one or more transactions involved in the dead-lock, they can be repaired. In this case, the scheduler can first pick a transaction T involved in the deadlock and release one or more locks acquired by T that participate in the deadlock. Then, the scheduler has to partially rollback T to a checkpoint before the released locks were acquired. Similar to MV3C, one should find a low-overhead automatic checkpointing mechanism to make such an approach practical.

Last but not least, we envision a system where the concurrency control algorithm itself is specialized and compiled based on the features that are necessary and sufficient for the database application code.

Bibliography

[1] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, June 2012.

[2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 241–252, New York, NY, USA, 2012. ACM.

[3] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL*, pages 293–302, 1989.

[4] J. Aspnes, A. Fekete, N. Lynch, M. Merritt, and W. Weihl. A theory of timestamp-based concurrency control for nested transactions. In *Fourteenth International Conference on Very Large Databases*, pages 431–444, Aug. 1988.

[5] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, Nov. 2014.

[6] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Trans. Database Syst.*, 5(2):139–156, June 1980.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment*, 2(2):1648–1653, 2009.

[9] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.

[10] V. Breazu-Tannen, P. Buneman, and L. Wong. *Naturally embedded query languages*. Springer, 1992.

[11] K. Brown, A. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100, 2011.

[12] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[13] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, Gensym, and reflection. In *GPCE*, 2003.

[14] M. J. Carey. Improving the performance of an optimistic concurrency control algorithm through timestamps and versions. *IEEE Trans. Softw. Eng.*, 13(6):746–751, June 1987.

[15] M. A. Casanova. *Concurrency Control Problem for Database Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1981.

[16] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *OOPSLA*, pages 835–847, 2010.

[17] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[18] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, SIGMOD '82, pages 184–191, New York, NY, USA, 1982. ACM.

[19] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: a hardness study and a principled heuristic solution. *IEEE Trans. Knowl. Data Eng.*, 2004.

[20] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. In *VLDB*, volume 97, pages 146–155, 1997.

[21] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. ICDE'11, pages 375–386. IEEE, 2011.

[22] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*. www.cidrdb.org, 2013.

[23] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, July 2012.

[24] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 931–942, New York, NY, USA, 2014. ACM.

[25] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. *IEEE Data Eng. Bull.*, 37(1):48–59, 2014.

[26] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM.

[27] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[28] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. *SIGPLAN Not.*, 42(9):315–326, Oct. 2007.

[29] D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting haskell strings. In *Proceedings of the 9th international conference on Practical Aspects of Declarative Languages*, PADL'07, pages 50–64, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware:" big" data, big analytics, small clusters. In *CIDR*, 2015.

[31] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[32] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, 2013.

[33] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.*, 7(4):277–288, Dec. 2013.

[34] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[35] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *J. Funct. Program.*, pages 455–481, May 2003.

[36] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, 2015.

[37] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, July 2015.

[38] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. *J. Comput. Syst. Sci.*, 41(1):65–156, Aug. 1990.

[39] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[40] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, Dec. 1987.

[41] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[42] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. FPCA, pages 223–232. ACM, 1993.

[43] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society.

[44] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[45] T. Grust and M. Mayr. A deep embedding of queries into Ruby. In *ICDE*, 2012.

[46] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry – database-supported program execution. SIGMOD '09, pages 1063–1066, New York, NY, USA, 2009. ACM.

[47] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe linq compilation. *Proceedings of the VLDB Endowment*, 3(1-2):162–172, 2010.

[48] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *ICDE*, 1997.

[49] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proc. VLDB Endow.*, 1(1):1107–1123, Aug. 2008.

[50] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[51] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 383–394, New York, NY, USA, 2005. ACM.

[52] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[53] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[54] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 international conference on Management of data, SIGMOD '10*, pages 603–614, New York, NY, USA, 2010. ACM.

[55] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[56] S. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008.

[57] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision locks. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 143–147, New York, NY, USA, 1981. ACM.

[58] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[59] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.*, 9(12):972–983, Aug. 2016.

[60] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *CIDR*, 2015.

[61] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[62] O. Kennedy, Y. Ahmad, and C. Koch. DBToaster: Agile views for a dynamic data management system. In *CIDR*, pages 284–295, 2011.

[63] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1675–1687, New York, NY, USA, 2016. ACM.

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

[65] C. Koch. Abstraction without regret in database systems building: a manifesto. *IEEE Data Eng. Bull.*, 37(1), 2014.

[66] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 613–624, 2010.

[67] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. Vaxcluster: A closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, May 1986.

[68] S. Kulkarni and J. Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *SIGPLAN Not.*, 47(10):147–162, Oct. 2012.

[69] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[70] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4), 1964.

[71] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, Dec. 2011.

[72] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[73] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, New York, NY, USA, 2014. ACM.

[74] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 27:1–27:14, New York, NY, USA, 2014. ACM.

[75] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic query transformations for dynamic web applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1175–1178, Washington, DC, USA, 2009. IEEE Computer Society.

[76] M. Mayr. *A Deep Embedding of Queries into Ruby*. PhD thesis, Universität Tübingen, 2013.

[77] E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *SIGMOD*, pages 706–706, 2006.

[78] T. Merrifield and J. Eriksson. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 127–139, New York, NY, USA, 2013. ACM.

[79] J. E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[80] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[81] D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic Optimization of Declarative Queries. PLDI '11, pages 121–131, New York, NY, USA, 2011. ACM.

[82] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. *Proc. VLDB Endow.*, 7(12):1095–1106, Aug. 2014.

[83] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[84] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, 2015.

[85] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, EPFL, 2004.

[86] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. ICDE'01, pages 567–574. IEEE, 2001.

[87] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939, Sept. 2010.

[88] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Inc., New York, NY, USA, 1986.

[89] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, Aug. 2012.

[90] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.

[91] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 133–144. ACM, 2012.

[92] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using jvm. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 23, Washington, DC, USA, 2006. IEEE Computer Society.

[93] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA, 1978.

[94] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, Feb. 1983.

[95] M. Reimer. Solving the phantom problem by predicative optimistic concurrency control. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, pages 81–88, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[96] A. Reuter. Concurrency on high-traffic data elements. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '82, pages 83–92, New York, NY, USA, 1982. ACM.

[97] K. F. D. Rietveld and H. A. G. Wijshoff. Reducing layered database applications to their essence through vertical integration. *ACM Trans. Database Syst.*, 40(3):18:1–18:39, Oct. 2015.

[98] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136, 2010.

[99] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 497–510, New York, NY, USA, 2013. ACM.

[100] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL*, pages 497–510, 2013.

[101] M. L. Scott. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009.

[102] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10:445–456, December 2016.

[103] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *SIGMOD*, 2016.

[104] N. Shamgunov. The memsql in-memory database system. In *IMDM@ VLDB*, 2014.

[105] X. Shi, B. Cui, G. Dobbie, and B. C. Ooi. Towards unified ad-hoc data processing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1263–1274, New York, NY, USA, 2014. ACM.

[106] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. SIGMOD '12, pages 731–742, New York, NY, USA, 2012. ACM.

[107] J. M. Smith and P. Y.-T. Chang. Optimizing the performance of a relational algebra database interface. *Commun. ACM*, 18(10):568–579, 1975.

[108] M. Stonebraker. Technical perspective - one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.

[109] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[110] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, Apr. 2014.

[111] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky. OptiML: an implicitly parallel domain-specific language for machine learning. In *ICML*, 2011.

[112] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM.

[113] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.

[114] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[115] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[116] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[117] P. Tözün, I. Atta, A. Ailamaki, and A. Moshovos. Addict: Advanced instruction chasing for transactions. *Proc. VLDB Endow.*, 7(14):1893–1904, Oct. 2014.

[118] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11. http://www.tpc.org/tpcc.

[119] Transaction Processing Performance Council. TPC-E Benchmark Revision 1.13.0. http://www.tpc.org/tpce/.

[120] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[121] P. Valduriez. *Shared-Disk Architecture*, pages 2637–2637. Springer US, Boston, MA, 2009.

[122] T. L. Veldhuizen. Incremental maintenance for leapfrog triejoin. *CoRR*, abs/1303.5313, 2013.

[123] T. L. Veldhuizen. Transaction repair: Full serializability without locks. *CoRR*, abs/1403.5645, 2014.

[124] S. Viglas, G. M. Bierman, and F. Nagel. Processing declarative queries through generating imperative code in managed runtimes. *IEEE Data Eng. Bull.*, 37(1):12–21, 2014.

[125] Y. Voronenko, F. Franchetti, F. de Mesmay, and M. Püschel. Generating high-performance general size linear transform libraries using Spiral. In *HPEC*, 2008.

[126] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, pages 344–358. Springer, 1988.

[127] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[128] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. POPL '07, pages 199–210, New York, NY, USA, 2007. ACM.

[129] B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. OOPSLA '08, pages 19–36, New York, NY, USA, 2008. ACM.

[130] A. Wolski. TATP Benchmark Description (Version 1.0). http://tatpbenchmark.sourceforge.net, Mar. 2009.

[131] Y. Wu. THEDB (Cavalia). https://github.com/Cavalia/Cavalia, 2016.

[132] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1689–1704, New York, NY, USA, 2016. ACM.

[133] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment*, 9(5):444–455, 2016.

[134] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[135] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, 2016. ACM.

[136] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1349–1350. IEEE, 2012.

List of Figures

List of Tables

Mohammad Dashti

✉ *Chemin des Noutes 17*
1023 Crissier, Switzerland
✉ *mohammad.dashti@epfl.ch*
+41 78 857 75 13
linkedin.com/in/mdashti
github.com/mdashti

Education

2017 **École Polytechnique Fédérale de Lausanne (EPFL)**, School of Computer and Communication Sciences, PhD. in the field of Database Systems, Switzerland. PhD thesis advisor: Prof. Christoph Koch

2012 **Sharif University of Technology**, Computer Engineering Department, MSc. in Information Technology Engineering, Tehran, Iran, GPA: 19.19/20. M.Sc. thesis advisor: Prof. Jafar Habibi

2010 **Sharif University of Technology**, Computer Engineering Department, BSc. in Information Technology Engineering, Tehran, Iran, GPA: 18.14/20

Core Experience

2012-2017 **DATA lab at EPFL**, Lausanne, Switzerland, Doctoral Assistant, and worked on the following projects:

- ✧ **DBToaster** (dbtoaster.org): an SQL-to-native-code compiler. It generates lightweight, specialized, embeddable query engines for applications that require real-time, low-latency data processing and monitoring capabilities. I am an active contributor to this project.
- ✧ **FS-Store**: I ***built*** a transaction processing system that exploits compilation techniques to optimize transaction programs for achieving the performance comparable to the hand-optimized low-level code.
- ✧ **Lightweight Modular Staging (LMS)** (github.com/epfldata/lms): a compilation framework that provides a library of core components for building high performance code generators and embedded compilers in Scala. I am a contributor to this open-source project.
- ✧ **Multi-Version Concurrency Control with Closures (MV3C)**: ***designed*** a novel concurrency control technique and ***implemented it in Scala and C++***. MV3C is best suited for high contention scenarios, as it has an efficient conflict resolution and transaction repair mechanism.

2016 **Microsoft Research**, Redmond, Washington, USA (3–month internship, May – August 2016)

Research intern in the Orleans Project, a framework that provides a straightforward approach to building distributed high-scale computing applications. Mentored by Dr. Phil Bernstein, ***designed and implemented the indexing and query functionality for project Orleans.***

2007-2012 **Sharif Processors Company**, Tehran, Iran

Co-founder of an enterprise software development company that analyzes, designs and implements information systems.

- ✪ **End-User Report Builder (EURB)** (github.com/mdashti/EURB): a web-based software tool for end-users to build and format their reports from their distributed databases. I am the main developer and maintainer of this project.

Publications

- ➢ Mohammad Dashti, Sachin Basil John, Thierry Coppey, Amir Shaikhha, Vojin Jovanovic, and Christoph Koch. "*Compiling Database Application Programs*". SIGMOD 2018, under submission.
- ➢ Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. "*Transaction Repair for Multi-Version Concurrency Control*". SIGMOD 2017.
- ➢ Philip A. Bernstein, Mohammad Dashti, Tim Kiefer, and David Maier. "*Indexing in an Actor-Oriented Database*". CIDR 2017.
- ➢ Milos Nikolic, Mohammad Dashti, and Christoph Koch. "*How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates*". SIGMOD 2016.
- ➢ Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. "*How to Architect a Query Compiler*". SIGMOD 2016.
- ➢ Aleksandar Vitorovic, Mohammed Elseidy, Khayyam Guliyev, Khue Vu Minh, Daniel Espino, Mohammad Dashti, Yannis Klonatos, and Christoph Koch. "*Squall: scalable real-time analytics*". VLDB 2016.
- ➢ Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, and Christoph Koch. "*Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems*". SNAPL 2015.
- ➢ Tiark Rompf, Nada Amin, Thierry Coppey, Mohammad Dashti, Manohar Jonnalagedda, Yannis Klonatos, and Martin Odersky, "*Abstraction Without Regret for Efficient Data Processing*". Data-Centric Programming Workshop 2014.

Technical Skills

Programming Languages	Java, Scala, C/C++, C# (.NET), JavaScript, HTML, CSS, SQL, PHP, Python
Web Dev	Spring, Hibernate, Symfony, jQuery, ExtJS
Databases	Relational: Oracle DB, MySQL, PostgreSQL / NoSQL: HBase, Microsoft DocumentDB
Big Data	Familiar with the Data Mining and Machine Learning techniques
Tools	Hadoop, Yarn, Spark, SVN, Git, Maven, LaTeX, JUnit, Apache Ant, JDBC, Cordova/PhoneGap

Honors

2012 Awarded **EPFL Graduate Fellowship** in recognition to my academic excellence.

2012 Recipient of the **Permission to Develop a Software Project as Mandatory Public Service** from Iran's "National Elites Foundation".

2010 Awarded **Exceptional Talent Fellowship** for the M.Sc. Program in Information Technology Engineering from Computer Engineering Department of Sharif University of Technology

.

2007 **First Place** in the first **Sharif Java Challenge**, Sharif University of Technology.

2006 **First Place** in the **National Skills Competition**, Major: Information Technology.

Additional Experience

2009-2012 **Sharif Enterprise Software**, Tehran, Iran

Worked as the **head JavaEE developer** on the following main projects:

- **Project.net**: an open source JavaEE project management software. I was ***the main developer*** in a private branch of this project that planned, developed, maintained and localized the project and handled the special requirements from commercial customers of the system. In addition, I developed several DevOps tools and automated scripts to facilitate the maintenance of the project.
- **General Automatic Application Updater (GAAU) (B.Sc. Project)**: a programmable tool for the automation of the installation and update processes of distributed software systems. I designed and implemented this tool. It was used in the customized Project.net project, which considerably decreased the maintenance costs..
- **Sharif Java Profiler (part of M.Sc. Project)**: a tool for extracting runtime information about Java programs including the accessed classes and method calls along with their frequencies. I ***designed and implemented*** this tool that was used in the customized Project.net project to assist the extraction of its software architecture.

2008-2009 **System Group Company**, Tehran, Iran

Worked as a **JavaEE developer** in the Java Enterprise Software Development team. I developed several main components of a CRM application in this team.

2007-2012 Worked on several side projects with affiliation to the **Sharif Processors Company**, Tehran, Iran:

- **Moragheb**: an e-Health national project for automating the processes related to controlling the health issues in Iran. This was my ***first commercial project*** that I developed by myself from scratch using JBoss Seam framework.
- **Network Map Editor (NME)**: a tool for documenting and simulating physical network layer of a large computer network, e.g., a country's telecommunication infrastructure. I designed and developed it.
- **Atash Firewall**: a personal firewall for MS Windows. Special features in this firewall are application window control, low-level network access, Network-enabled application launch, and Detecting a change in an executable. I was a ***key developer and designer*** in this project.
- **SIAMRA**: a web-based Information System for Iran Computer Museum. It automated the internal processes of the museum as well as serving the clients. I was the ***main developer*** in this project.
- **SPY Task Manager**: a task manager for small and medium-sized businesses that I ***built*** from scratch.
- **DForum**: an online discussion forum written in PHP Symfony framework. I was a ***key developer***.

2012-2017 Was a teacher assistant in **Big Data**, **Software Engineering**, **Functional Programming** and **IT Security** courses with more than 120 students in each course.

Speaking Languages

114 English: Fluent, C2 equivalent / Persian: Mother Tongue / French: Basic proficiency, A2 equivalent