A BARE-METAL APPROACH TO DATABASE PERFORMANCE ON CONTEMPORARY HARDWARE

by

Craig Chasseur

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 9/8/15

The dissertation is approved by the following members of the Final Oral Committee:
Jignesh M. Patel, Professor, Computer Sciences
Jeffrey Naughton, Professor, Computer Sciences
Anhai Doan, Professor, Computer Sciences
Remzi Arpaci-Dusseau, Professor, Computer Sciences
Terry Millar, Professor Emeritus, Mathematics

In loving memory of my grandfather, Robert P. Rohde.

ACKNOWLEDGMENTS

None of the research described in this thesis would have been possible without a team of truly excellent collaborators. First and foremost among these is Professor Jignesh M. Patel, who as an advisor has given excellent guidance and insight, along with a great deal of encouragement and patience. Professor Patel's vision created the Quickstep project and made it the success it is today, and the author is forever grateful for the opportunity to work on such an exciting project and to have such an excellent mentor.

A great deal of credit is due to the other graduate students who worked with the author on the Quickstep project, in particular Yinan Li (who designed BitWeaving and WideTable), Harshad Deshmukh (who developed the Quickstep scheduler), Qiang Zeng (who rewrote the Quickstep query optimizer), Shoban Chandrabose (who developed most of the aggregation and sorting features in Quickstep), Zuyu Zhang (who implemented some key types for TPC-H and has done key work on the distributed version of Quickstep), Adalbert Gerald Soosai Raj (who developed the NUMA partitioning features in Quickstep), James Paton (who wrote the Quickstep buffer manager), and Sangmin Shin (who developed the original version of Quickstep's aggregation code). Many of these students are also coauthors on a paper which Chapters 2 and 5 of this thesis are adapted from. This thesis would not have been possible without their efforts.

Thanks is also due to other graduate students who have participated in the design process for Quickstep and contributed valuable ideas and feedback, as well as volunteering to edit papers and give advice. The author particularly wishes to thank Spyros Blanas, Avrilia Floratou, Jing Fan, and Navneet Potti.

The author benefited immensely from two internships at Google during his studies. The experience gained there and lessons learned had a great impact in making Quickstep a more disciplined and well-engineered project. Special thanks is due to Navin Melville, John Cieslewicz, and Himani Apte, who served as fantastic hosts and mentors during these internships. The author was also supported by a Google PhD fellowship during the final year of his studies.

Finally, the author would like to thank his family for their unwavering support and love throughout his studies and life. Special thanks go to his parents, Mary and Tim Chasseur for their unconditional love and constant encouragement, to his cousin Kathryn Brownell (nee Cramer) for valuable advice on life as an academic, and to his late grandfather Robert P. Rohde who fostered a lifelong love of science and engineering.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

A BARE-METAL APPROACH TO DATABASE PERFORMANCE ON CONTEMPORARY HARDWARE

Craig Chasseur

Under the supervision of Professor Jignesh M. Patel
At the University of Wisconsin-Madison

Main-memory analytical databases have experienced a major surge of interest in recent years, driven by increasing DRAM density and dropping prices that have made in-memory analytics practical for many new workloads. Main-memory DBMSes have realized signficant performance improvements over their disk-based counterparts, but many existing systems still employ data processing methods (e.g. the implementation of core relational operations like selection and joins) that were developed for a now-bygone hardware era. This dissertation describes research aimed at closing the gap between the bare-metal performance that modern hardware is capable of and the query-processing performance that is actually achieved by analytics platforms. The approach taken is to develop a working prototype DBMS, called Quickstep, as a platform for experiments. Quickstep was designed from scratch to take advantage of modern hardware capabilities, particularly large main memories (often in NUMA configuration), fast on-die caches, and CPUs that expose a great deal of parallelism both in terms of multiple cores and instruction-level parallelism. Quickstep employs a unique, flexible block-oriented storage design, and a parallel and elastically scalable query scheduling and execution framework. The design of the complete Quickstep engine is described in this dissertation, and various empirical experiments are carried out to evaluate important aspects of the design. Key components of Quickstep, the storage engine and the communication framework (called the *Transactional Message Bus* or *TMB*), are covered in especially great detail. Experimental findings, in single-node settings, regarding the in-memory performance and cache behavior of various storage organizations, as well as a detailed discussion and definition of messaging semantics and their implementation in a reusable TMB library, are presented and are broadly applicable beyond the Quickstep project.

Jignesh M. Patel

ABSTRACT

Main-memory analytical databases have experienced a major surge of interest in recent years, driven by increasing DRAM density and dropping prices that have made in-memory analytics practical for many new workloads. Main-memory DBMSes have realized significant performance improvements over their disk-based counterparts, but many existing systems still employ data processing methods (e.g. the implementation of core relational operations like selection and joins) that were developed for a now-bygone hardware era. This dissertation describes research aimed at closing the gap between the bare-metal performance that modern hardware is capable of and the query-processing performance that is actually achieved by analytics platforms. The approach taken is to develop a working prototype DBMS, called Quickstep, as a platform for experiments. Quickstep was designed from scratch to take advantage of modern hardware capabilities, particularly large main memories (often in NUMA configuration), fast on-die caches, and CPUs that expose a great deal of parallelism both in terms of multiple cores and instruction-level parallelism. Quickstep employs a unique, flexible block-oriented storage design, and a parallel and elastically scalable query scheduling and execution framework. The design of the complete Quickstep engine is described in this dissertation, and various empirical experiments are carried out to evaluate important aspects of the design. Key components of Quickstep, the storage engine and the communication framework (called the *Transactional Message Bus* or *TMB*), are covered in especially great detail. Experimental findings, in single-node settings, regarding the in-memory performance and cache behavior of various storage organizations, as well as a detailed discussion and definition of messaging semantics and their implementation in a reusable TMB library, are presented and are broadly applicable beyond the Quickstep project.

1 INTRODUCTION

The present moment in the evolution of analytic database systems is one of incredible possibilities, but also of difficult challenges. Hardware trends, particularly increasing DRAM density and dropping prices, have led to a resurgent interest in main-memory databases that can run complex queries on large datasets much more quickly than disk-based systems. Despite the impressive performance that main-memory based analytics systems have achieved relative to "legacy" database engines, the various systems in this category still, to a greater or lesser degree, embody design decisions that are based in conventional wisdom from a bygone hardware era (e.g. file-based storage layouts optimized for disk I/O, or parallel execution models that assume "shared-nothing" even though today's multicore systems are shared-memory). This thesis demonstrates how redesigning key analytical database components like the storage engine, the execution framework, and the communication layer based on the actual capabilities and limitations of modern and emerging hardware (i.e. thinking from the bare-metal up) can allow further significant performance improvements to be realized in main-memory analytical databases.

Performance improvements resulting from hardware-aware database design will be crucial in meeting the increasing demands that are placed on in-memory databases. These demands are not merely an issue of growing data sets that can be solved by buying more, denser memory and storage devices, but the complexity of workloads and the need for low-latency query response times are also rapidly increasing. An in-memory analytical database is often the underlying data management technology used by statistical modeling or machine-learning software, and such applications can issue many complex queries in the course of their normal operations. There is also a need for fast "interactive" response times to accomodate the needs of users, whether they are analysts and data scientists exploring a data warehouse to extract insights and value, or end users of a data-driven application.

The platform for the research described in this thesis is a new working prototype DBMS called Quickstep. Quickstep was developed to serve as a high-quality real-world software testbed to conduct practical experiments about efficient data processing on a modern server. The focus is mainly on the single-node setting where data sets fit in main memory, although the work described is for the most part also applicable to query processing on individual nodes in a distributed database, and design choices have been made with the intent of developing a distributed version of Quickstep in the future. Quickstep was de-

veloped from scratch at the University of Wisconsin, and the thesis author has been one of the primary developers since the project's inception.

The first major research undertaking with Quickstep was an in-depth empirical evaluation of different main-memory storage organizations for data. The Quickstep storage manager is designed to support flexibility in terms of the physical layout of data along several degrees of freedom, allowing different tuple-storage layouts (e.g. row-store, column-store, or other possible designs like PAX or Data Morphing), different secondary index structures (e.g. cache-sensitive B+-Trees, main-memory hash tables, etc.), and optional features like compression, most of which can be freely used in combination with each other for a truly "mix and match" storage architecture. The key to Quickstep's flexible storage organization is the concept of the *Storage Block*, a self-contained, self-describing unit of storage for a segment of a data table whose internal organization is opaque to the rest of the query processing system and supports an API consisting of simple, logically-described relational operations like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. A block decides for itself based on its own internal organization how to most efficiently perform these operations, masking the complexity of handling many different possible storage formats from the rest of the system. This flexibility allowed one of the first truly apples-to-apples performance comparisons of different storage organizations for read-optimized main memory databases to be carried out, which explored the design space of main memory data storage in a single node along four dimensions: blocks vs. conventional files, row-stores vs. column-stores, the use of indexing, and the use of compression. These experiments showed that Quickstep's novel block-based organization outperformed the traditional file-based organization used by most legacy databases, and showed that different options for tuple layout, indexing, and compression all had their own performance "sweet spots" depending on data and query parameters. Chapter 3 covers these experiments in detail.

The next effort with Quickstep was to develop a communications framework that would meet both the scalability and reliability needs of the project. Communication is at the heart of any parallel or distributed application, and Quickstep is no different. Workers in Quickstep need to be able to exchange messages with each other quickly and reliably in order for the system to "scale up" to multi-socket NUMA servers with many CPU cores and to "scale out" to distributed clusters of machines [1]. A new communication paradigm

[1]This thesis is largely concerned with Quickstep as a single node system, but work is already underway on development of a distributed version of Quickstep. The TMB is key to this work, as it provides the same network-transparent messaging services for Quickstep components both within a process and across a cluster.

called the *Transactional Message Bus* (TMB for short) was developed to address these needs. Realizing that the need for a robust communication subsystem with both intra-node and inter-node scalability extends far beyond just the Quickstep project, TMBs were developed as a self-contained library that can be released and used separately in other applications. Briefly, a TMB is a message bus that provides clean, well-defined semantics for sending messages between actors in a parallel or distributed system. TMBs are *transactional* in that the act of sending or receiving a message is an ACID transaction with guaranteed delivery, data persistence and recovery support, and a consistent, deterministic set of semantics for addressing and ordering messages. Several full-fledged TMB implementations were developed based on existing SQL and NoSQL data processing engines (including SQLite, VoltDB, LevelDB, and Apache Zookeeper), as well as a custom transaction manager, write-ahead logging mechanism, and network protocol that were written from scratch. A performance evaluation of these different TMB implementations was conducted, measuring both their intra-node scalability in a high-end multi-socket NUMA server and their inter-node scalability running on many servers in a cluster or cloud environment. The functionality and performance of TMBs was also compared with existing message-oriented middleware. The TMB paradigm and these experiments are discussed in detail in Chapter 4.

Finally, with these core components (the Quickstep storage manager and the TMB) in place, Quickstep was developed into a complete SQL analytics engine. This involved the development of a multithreaded execution engine that uses storage blocks as a natural unit of parallelism and data flow, and uses the TMB to coordinate inter-thread communication between the scheduler and workers. This also required the development of relational operator implementations that fit in with this parallel and elastic execution model. For more complex operators like hash join and aggregation with `GROUP BY`, latch-free data structures are used for shared state to minimize synchronization overhead and allow even these more complex operators to scale well with additional resources. Members of the Quickstep development team collaborated to integrate features like NUMA-aware data partitioning and "WideTable" [71] schema denormalization and query transformations with the Quickstep engine. The overall design of the Quickstep system is presented in Chapter 2, and empirical experiments illustrating important properties of the holistic system are presented in Chapter 5.

The rest of this dissertation is organized as follows. Chapter 2 discusses the design and engineering of Quickstep at a high level. Chapter 3 details the design and implementation

of the Quickstep storage engine in greater detail, including the novel block-oriented architecture. Chapter 3 also presents results from experiments measuring the performance of different storage organizations in main memory, demonstrating above all the utility of a flexible storage system that allows for many different data organization options. Chapter 4 presents the Transactional Message Bus, a scalable and reliable communications paradigm that was developed for Quickstep, but has broader applications for other systems. Chapter 4 develops the abstract semantics of the TMB in detail, then discusses experiences in developing a number of compatible TMB implementations built on different underlying database technologies and presents an experimental evaluation of the performance and scalability characteristics of these different implementations, including a comparison with existing message-oriented middleware. Chapter 5 presents results from experiments on end-to-end query processing in Quickstep. Chapter 6 contains concluding remarks.

2.1 Introduction

This chapter describes the overall design of the Quickstep database management system. Quickstep has been designed with the aim of exploiting important trends in server hardware and closing the gap between the "bare-metal" capabilities of the underlying hardware and the performance that is actually delivered by main-memory analytical databases. The particular hardware trends that are of interest include large main memory capacities (which have motivated the recent surge in interest in main-memory DBMSes in the first place), fast on-die CPU caches, parallel multi-core CPUs with steadily increasing core counts in each hardware generation, and CPU cores that are capable of instruction-level parallelism exposed in the form of SIMD (single-instruction multiple-data) instruction sets.

A natural question is whether the technological factors listed above require a rethinking of how we build modern analytical systems. Over the last five years Quickstep has served as an experimental platform to explore these issues. Cumulative findings from the Quickstep project include the following:

First, the Quickstep storage manager (discussed in detail in Chapter 3) has a unique block layout, where each block behaves like a mini self-contained database. Thus, any indices are packed in the block along with the actual tuple data. This design allows Quickstep to easily transform data locally within a block to best suit the organization that maximizes query performance, and reduces the need for global coordination for physical schema changes. For example, data that is being loaded can be stored in a block that is a row-store, and that block can be morphed into a compressed row store with BitWeaved indices when it is full to speed up subsequent analytics queries. The Quickstep block-based approach to storage management also naturally leads to a highly parallelizable query execution paradigm in which independent work orders are generated at the block level. Query execution then becomes a matter of creating and scheduling work orders, which can be done in a generic way. This method of "breaking down" the query into work orders at the block-level implies that system management aspects, such as dealing with elasticity mid-flight through query execution, query progress monitoring, and dealing with stragglers, become easier to handle (as it is done in one place – i.e. the scheduler, and in a generic way across all queries). Thus, the query scheduler is a critical component of the

Quickstep architecture.

Second, Quickstep uses novel query processing techniques for high performance. The scan/select operations use BitWeaved indices [70] (whenever possible) that are efficient for scans. Quickstep also implements other relational operators, including hash-based joins and hash-based aggregation, and uses state-of-the-art parallel latch-free algorithms for these operators. In addition, a database in Quickstep can be "denormalized" using WideTables [71], which examine the schema to flatten out the database into a set of denormalized tables that are materialized. With this approach, a large class of complex queries can be converted to (fast) scans on the WideTables; queries that can't be transformed are evaluated in the "traditional" way by creating, scheduling, and executing a query plan with the usual relational operations.

Finally, while Quickstep can work with data that does not fit in memory (it has an LRU-k based buffer manager), it is really optimized for the case when the dataset fits in main memory. Thus, it aims to leverage the in-memory technological trend to produce a data analytics system that executes queries fast.

2.2 Quickstep Architecture

The logical view of the Quickstep architecture is that it implements a collection of relational algebraic operations, using efficient algorithms for each (more details are discussed in Section 2.4). This kernel can be used to run a variety of applications that can be run on top of a relational query processing engine. These applications include traditional SQL Analytics (a.k.a. Data Warehousing), but can also include other analytical application classes including Graph analytics and relational learning as programs in these application classes can be mapped to an underlying relational algebra [34, 56, 111]. The focus of this chapter is on the SQL analytics component.

Data Model and Query Language

Quickstep uses the standard relational data model, and SQL as the query language. Currently, the system supports the following basic types: `INTEGER` (32-bit signed), `BIGINT/LONG` (64-bit signed), `REAL/FLOAT` (IEEE 754 *binary32* format), `DOUBLE PRECISION` (IEEE 754 *binary64* format), fixed-point `DECIMAL`, fixed-length `CHAR` strings, variable-length `VARCHAR` strings, `DATETIME/TIMESTAMP` (with microsecond resolution), date-time `INTERVAL`, and year-month `INTERVAL`. The type system code is internally organized as a (C++) hierarchy, allow-

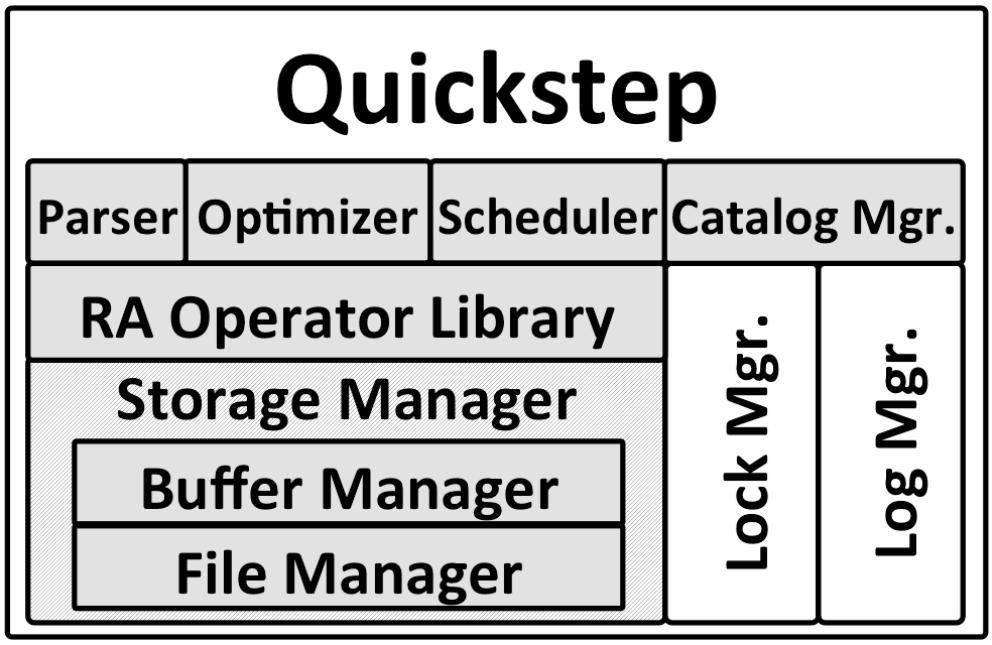


Figure 2.1: The Quickstep block-level architecture.

ing one to easily add in new types to the system. However, at this point adding in new types requires recompilation of the system. Dynamic types or user-defined types are not yet supported in Quickstep, but the type system has been designed to allow for such additions in the future.

At this point, Quickstep supports a limited class of SQL, which includes correlated queries, but does not include more advanced SQL, such as window functions.

System Overview

The block-level diagram of the components in Quickstep is shown in Figure 2.1.

The system has a SQL *parser* that converts the input SQL query into a syntax tree. This syntax tree is then sent to an optimizer that converts the syntax tree to a physical plan. The *optimizer* first converts the syntax tree to a logical plan, and uses a traditional rules-based transformation-based approach [42] to convert the logical plan to an optimal physical plan. The current optimizer is simple (but extensible) and supports projection and selection push-down, and join-order optimization. Only left-deep trees are currently considered.

The *catalog manager* keeps track of the logical and physical schemas of the database. The catalogs keep simple statistics at this point, including estimated table cardinalities. More sophisticated statistics such as histograms are planned for addition in the future.

The catalog manager supports an API to write all the catalogs as a *protobuf* to allow exporting the schema for external purposes (e.g. sending the schema to a modular stand-alone optimizer like Orca [93]).

Physical plans created by the optimizer are then handed over to the *scheduler*. The physical plan is a directed acyclic graph (DAG) of relational operators. The current *relational operator library* contains the implementation of various relational operators including selection, projection, joins, aggregation, sorting, and top-k. Additional details about the operators are provided in Section 2.4.

The *storage manager* uses a unique block-based design that organizes the data into large multi-MB blocks. Each block contains tuples from a single table, and is treated as a "mini-database." Different blocks, even within the same table, may have different physical organizations. The external view of each block is that of a bag of tuples, and query processing simply invokes set-oriented methods on each block. This design allows for the physical schema for each block to evolve independently. There are no global indices in Quickstep; instead indices are self-contained within blocks. Different block formats are supported including column stores and row stores (see Chapter 3 for more details) along with BitWeaving and CSB+-Tree indices. This unique storage manager design implies that tuples in one block for a table may be organized as a column-store (as queries on that block may involve selecting only a few attributes), while another block for the same table may be organized as a row-store (with perhaps CSB+-Tree indices). Each block can be optimized for its local pattern of access, which can be recorded succinctly within each block, and the block's physical layout can be *morphed* to adapt to its actual pattern of access [47]. (Data morphing techniques are not yet implemented in Quickstep and that is part of planned future work; for now, Quickstep provides a tool that can specify the physical layout of each block using user-supplied hints.) It is natural to wonder whether such a flexible storage system complicates the optimizer, particularly the question of access path selection. Quickstep addresses this issue by removing the responsibility of low-level physical access path selection from the global query optimizer and embedding it in the individual blocks, which locally decide for themselves based on their own physical organization how to implement the simple relational operations like predicate evaluation and projection exposed to operators by the storage block API. Thus, a scan on one block may involve scanning all the columns in a column-store block, and the scan on another block that has an index on the scan predicate may end up using the index. Simple statistics on attributes are kept within each block to aid in this run-time decision.

The *scheduler* is another unique component of the system. The scheduler breaks up a query into various *work orders* and query execution is essentially a series of work order generation and completion tasks. Details about the scheduler are presented in Section 2.4.

2.3 Storage Management in Quickstep

The Quickstep storage manager is based on a novel block-based architecture that allows a large variety of different physical data organizations to coexist within the same database, and even within the same table. This flexible and extensible design for physical storage was chosen based on empirical experiments that show that the performance of core data-intensive relational operations like selection and projection can be sensitive to the layout of data in memory, and there is no single "one size fits all" storage organization that achieves good performance for all workloads. These experiments are described in detail in Chapter 3.

Storage for a particular table in Quickstep is divided into many blocks, with individual tuples wholly contained in a single block. Blocks of different sizes are supported, and based on the results described in Chapter 3, blocks that are about 16 megabytes in size are used as the default for current large memory systems. On systems that support large virtual-memory pages, block sizes are constrained to be an exact multiple of the hardware large-page size (e.g. 2 megabytes on x86-64) so that buffer pool memory can be allocated using large pages.

The Quickstep storage manager maintains a buffer pool of memory that is used to create blocks and to load them from persistent storage on-demand. Large allocations of unstructured memory can also be made from this same buffer pool, and are used for shared run-time data structures like hash tables for joins and aggregation operations. Thus, there is a single place where memory requirements for the query-processing engine are managed, and there is a holistic view of memory management. Slots in the buffer pool (either blocks or unstructured "blobs") are treated much like a larger-sized version of page slots in a conventional DBMS buffer pool, and there is a mechanism where different "pluggable" eviction policies can be activated to choose how and when blocks are evicted from memory and (if necessary) written back to persistent storage in accordance with user-specified limits on memory usage. Currently, the default eviction policy is LRU-2, but a "random" policy and LRU-k with arbitrary k are also supported, and the pluggable eviction policy system allows implementing other strategies easily in the future.

Data from the storage manager is backed by persistent storage through a file manager abstraction that currently supports POSIX file systems, Windows file systems, and also HDFS [10].

Block-Structured Storage

Internally, a block consists of a small metadata header (the block's self-description), a single *tuple-storage sub-block* and any number of *index sub-blocks*, all packed in the block's contiguous memory space. There are multiple implementations of both types of sub-block available in Quickstep (several are described below), and the API for sub-blocks is generic and extensible, making it easy to add more. When a tuple is inserted into a block, all column values are stored in the tuple-storage sub-block, and any applicable index sub-blocks are updated (index sub-blocks always refer to tuples in the same block, and thus indices are totally self-contained at the block level and always colocated with data).

In general, higher-level operator code need not concern itself with the internal structure of different storage blocks. Instead, blocks provide a common API consisting of a simple set of logical, relational operations including select/project (materializing output to other in-memory blocks), tuple insertion (both tuple-at-a-time and batch-oriented), in-place updates and deletes (with optional predicates), and sorting. The actual implementation of these operations may be different depending on what sub-blocks are present in a particular block, and each block decides for itself, based on its own internal organization, how to most efficiently execute a particular call.

As an example, a select operation may first select tuples based on a complex predicate tree involving conjunctions and disjunctions of leaf comparison predicates. The block's *micro-optimizer* gets a cost estimate of how expensive it will be to evaluate each leaf predicate using each sub-block (both the tuple-store and all the indices), and chooses the lowest-cost block to evaluate each individual leaf predicate. The set of matches for a predicate in a particular block is represented as a compact bit-vector. Arbitrary predicate trees are evaluated by simply performing bit-wise operations on the bit-vectors of matches (a "filter" bit-vector can also be passed in when evaluating some predicates so that it is possible to skip over or short-circuit evaluate some predicates within a conjunction or disjunction that has already been decided for some tuples).

Tuple Stores

Quickstep currently implements both row-store and column-store layouts for *tuple-storage sub-blocks*, although other formats (e.g. PAX [4]) are also possible. A detailed analysis of the in-memory performance characteristics of different options for tuple storage is presented in Chapter 3.

Row Stores

Row stores come in two flavors. The first is a "split" row-store that uses a conventional slotted-page layout with storage for fixed length attributes growing from one end of the sub-block memory, and storage for variable-length attributes growing from the other end. The second variety of row-store is a "packed" row store which is simply a packed row-major array of fixed-length values, and can not be used with relations that have variable-length attributes.

Column Stores

A column store tuple-storage sub-block divides sub-block memory into contiguous stripes, one for each column, and stores column values packed within the individual stripes. A column store may optionally be kept sorted on the values of one column.

Compression

Both row store and column store tuple-storage sub-blocks may optionally be used with compression. Quickstep implements simple ordered dictionary compression, with dictionaries constructed on a per-block basis and, naturally, self-contained within the block. Dictionary compression converts native column values into short integer codes that compare in the same order as the original values. Depending on the cardinality of values in a particular column within a particular block, such codes may require considerably less storage space than the original values. In a row store, compressed attributes require only 1, 2, or 4 bytes in a single tuple slot. In a column store, an entire column stripe consists only of tightly-packed compressed codes.

Because the implementation of dictionary compression is order-preserving, comparison predicates can be evaluated directly on the compressed codes without decompressing. This means that considerably less memory bandwidth and cache space may be used when scanning a compressed column (especially with a column store). Additionally, comparing

integer codes requires only a simple single-cycle CPU instruction, while comparing some more complex uncompressed data types (e.g. strings) can be considerably more CPU-intensive.

On the other hand, compression does introduce an additional level of indirection when native values have to be accessed by looking them up in a dictionary (e.g. when doing projection), so whether compression is valuable for a particular query workload is dependent on selectivity.

Indices

Quickstep currently implements several index sub-blocks, including CSB+-Tree [87], BitWeaving/H, and BitWeaving/V [70]. All index sub-blocks support the same interface allowing parts of a predicate tree to be evaluated using different sub-blocks, with a bit-vector of matching tuple positions as the common representation of predicate matches.

CSB+-Tree

The CSB+-Tree (cache-sensitive B+-Tree) [87] is a modification of the classic B+-Tree ordered index structure designed to achieve good performance in main-memory. Nodes in a CSB+-Tree are physically aligned with cache lines, and are allocated in contiguous "node groups" so that scanning across sibling nodes has a linear prefetching-friendly memory access pattern. CSB+-Trees are generally useful for highly-selective lookup queries.

BitWeaving

BitWeaving/H and BitWeaving/V are bit-based indexing methods [70], that aim to exploit the bit-level parallelism in the ALUs of modern CPUs. Briefly, BitWeaving/H (horizontal) improves on standard bit-packing for compressed codes by adding a "padding bit" next to each code packed into a word (using a technique originally proposed for computing byte-level arithmetic operations on parallel on a CPU with a wider word size [66]), and using a sequence of bitwise operations on the codes in a word that sets the value of the padding bit to 1 or 0 depending on whether a comparison condition matches for the respective code. Codes are organized into groups of words so that a word-long bit-vector of matches can be efficiently constructed by masking and shifting the padding bits. BitWeaving/V (vertical) decomposes the bits of a particular code vertically, packing the most-significant bits of several codes together into one word, the second-most-significant bits into the next

word, and so on. Comparisons are evaluated bit-by-bit on all the codes in parallel (e.g. on 64 codes in parallel on a typical CPU with 64-bit word size, or 128-512 codes when using SIMD registers). It is often possible to use "early pruning" to skip over some words containing less-significant bits when a particular comparison can be totally decided for a group of codes by only looking at some of the most-significant bits. The probability of early-pruning is increased when a filter from some other part of a conjunctive predicate is available. Experimental evaluation of BitWeaving techniques has shown that, for typical code lengths, the average number of CPU cycles/tuple required to evaluate a scan predicate is less than 1, achieving true intra-cycle parallelism [70].

2.4 Quickstep Query Execution

Query execution in Quickstep takes advantage of the unique block-based design that is employed by the storage manager (introduced in Section 2.3 and covered in more detail in Chapter 3) and uses a workflow-based query execution paradigm, in which query execution involves generating a series of *work orders* that are then executed independently by workers. A work order largely corresponds to applying some operation on a block of input tuples.

Operator Algorithms

This section briefly describes the operator algorithms that are currently implemented in Quickstep.

Selection

Quickstep has a standard implementation of the selection operator. The key difference is that selection operator can decide on a block-by-block basis as to what algorithm(s) to use to apply the selection operation. When selecting with a predicate, the "micro-optimizer" (see Section 2.3) may choose to evaluate some or all of the predicate using a fast-path like an index lookup (using a BitWeaving or CSB+-Tree index), or a binary search on a sorted column in a column-store. If no fast-path is available, it falls back to iterating over the tuples in a block and evaluating the predicate for each. The expressions in the projection list are then materialized, and the resultant tuples are bulk-inserted into a temporary in-memory block for consumption by other operators. Both scan-based predicate evaluation

and general expression evaluation benefit from vectorization, which is described below. In the common case when the projection simply copies column-values as-is from the input, expression evaluation is skipped entirely and data is directly copied from one block into another.

Join

A hash join algorithm has been implemented consisting of a build phase and a probe phase. These phases are implemented as separate operators. The build hash table operator reads blocks of the build relation, and builds a hash table in memory using the join predicate as the key. The probe hash table operator reads blocks of the probe relation, probes the hash table, and materializes joined tuples into in-memory blocks for consumption by other operators, just like selection does. Both operators naturally take advantage of block-level parallelism, and a latch-free concurrent hash table is used to allow many workers to proceed at the same time. For non-equijoins, a block-nested loops join operator has also been implemented (a "residual" filter predicate can also be used with a hash-join to filter joined tuple pairs on a non-equijoin predicate in addition to the equality predicate evaluated by hashing).

Aggregation

For aggregation without GROUP BY, the operator reads blocks of the input relation and generates work orders per block that compute the aggregates local to their assigned block and merge them with the global aggregates for the query. For aggregation with GROUP BY, the operator generates work orders per block that together build a hash table of aggregation handles in parallel using the grouping columns as the key (again, a latch-free hash table is used here). After processing all the blocks in the input relation, a work order iterates through the hash table to output the grouping keys and their corresponding aggregates, or just the aggregates in the case of aggregation without GROUP BY.

Sorting

A simple two phase algorithm was implemented for sorting and top-K. These phases are implemented as separate operators. In the first phase, each block of the input relation is sorted in-place, or copied to a single temporary sorted block. This phase is similar to the

run generation phase of the traditional external sort. In the second phase, runs of sorted blocks are merged to create a fully sorted output relation.

Vectorization

The work order execution model and dynamic scheduler in Quickstep are designed to take advantage of hardware parallelism in the form of multicore CPUs. Instruction-level parallelism is another level of parallelism *within* a CPU core that Quickstep also takes advantage of by means of vectorization.

Modern server CPUs have SIMD (single-instruction multiple-data) instruction sets that allow an arithmetic operation to be applied to vectors consisting of several elements of the same data type in a single instruction. Compilers implement vectorization passes as part of optimization and code generation, and these passes are able to transform scalar loops that apply the same operations to many data items serially into a faster vectorized form that uses SIMD instructions to process multiple data items in each loop iteration. Relying on the compiler avoids the chore of hand-writing SIMD assembly, and also has the advantage of portability across different instruction sets. However, not any loop can be vectorized. In particular, branching within the inner loop will confound vectorization (since different code-paths are taken on an element-by-element basis), as will most common forms of indirection like calling a function pointer or virtual method (or, for that matter, any function call that can not be inlined). The key to compiler-assisted vectorization, then, is to have simple, tight inner loops without branching or indirection.

Quickstep has a system of code templates for accessing column values of a particular type in a particular kind of tuple-storage sub-block (a *data-access template*). There are also code templates for evaluating different comparison predicates and simple scalar expressions with arguments of the various built-in SQL data types in Quickstep (an *expression template*). Combining an expression template with a data access template produces a function that evaluates an expression over all the tuples in a particular kind of tuple-storage sub-block (optionally filtered by a bitmap indicating matches for a previously-applied predicate). The functions produced have a simple inner loop with no branching or indirection, and are thus amenable to vectorization [1]. These functions produce output a column-at-a-time in temporary in-memory arrays that other expressions can also process

[1]This is true for integers (including compressed codes) and floating-point numbers that are supported by the SIMD instructions on a particular CPU. Other data types like uncompressed strings do not benefit from this technique.

in a vectorized fashion. So, even though there are only expression templates for simple arithmetic operations, arbitrarily complex expressions can be evaluated by doing a depth-first traversal of an arbitrary expression tree and doing column-at-a-time vectorized evaluation for each simple-expression node. The cross-product of data-access templates and expression-evaluation templates is pre-compiled into Quickstep so that the appropriate vectorized function can be selected at runtime.

Other operations like data movement between blocks, building and probing hash-tables for hash-joins, aggregation over a block (both simple and hash-based), and sorting also use the data-access templates so that data-intensive tuple-by-tuple inner loops are as simple as possible, even though SIMD isn't necessarily applicable in these situations.

Threading Model

The Quickstep execution engine uses two kinds of execution abstractions (spun up as threads), namely a *foreman* and *workers*. The foreman and workers all communicate via a *Transactional Message Bus* (described in detail in Chapter 4), which provides a reliable and network-transparent communication abstraction for the Quickstep components. (At this point Quickstep uses a single process with an in-process TMB instance for the database engine, but the design allows for a future version of Quickstep that has multiple processes, potentially spread across multiple machines.)

Worker threads execute work orders produced by physical relational operators. The foreman thread makes decisions about scheduling the work orders to the worker threads. The current execution engine allows a single query to be controlled by a single foreman thread. All worker threads are stateless. They simply keep executing work orders in a loop. A worker thread can be terminated by sending a special poison work order. The work order termination method can then be used to gracefully terminate the Quickstep process, or to abort a query (for example, in response to the client issuing a query cancellation command).

To minimize the thread initialization costs, all the worker threads are created when the Quickstep process is started, and they stay alive until the Quickstep process terminates. Note, however, that since workers are stateless, it is easy to add or remove them, even mid-query. This property of "instant elasticity" is illustrated by a preliminary experiment in Section 5.4.

A foreman thread is spawned for every new query that arrives to the system. A load controller limits the number of concurrent queries, but this component is rudimentary at

this point, and it is not discussed further here.

Work Order-based Scheduler

Query execution in Quickstep is based on a scheduler that breaks up the work for the entire query into a series of *work orders*. This section describes this work order-based scheduler.

Work Order Generation

A physical query plan in Quickstep system is represented as a directed acyclic graph (DAG) in which each node is a relational operator. Each relational operator presents the work that needs to be done to execute the query using *work orders*. The following example query is used to illustrate query execution using work orders:

```
SELECT SUM (sales)
FROM Product P NATURAL JOIN Buys B
WHERE B.date = 'June-30-2015'
  AND P.category = 'swim'
```

The optimal query plan that is produced by the optimizer is converted to a DAG of operators. This DAG representation of the query is then sent to the foreman for the query, which is responsible for scheduling the work that makes up the query. The DAG for the query above is shown in Figure 2.2. The edges of the DAG are annotated with whether the consumer operator is blocked on the output produced by the producer/upstream operator, or whether data pipelining is allowed between two neighboring operators. The foreman goes over the query DAG and fetches all the "schedulable" work orders from all the operators in the DAG. Initially, only the select relational operators (shown in the DAG using the symbol σ) generate work orders – one for each input block from both input relations.

Each selection work order contains the following information: a globally unique input block ID, input relation, filtering predicate, and the list of attributes (or general expressions) to be projected. The foreman schedules these work orders on to the available worker threads by sending messages containing a serialized representation of the work order to the worker over the TMB. The worker threads execute these work orders by carrying out the following steps: reading the input block, evaluating the predicate over the block, and writing its output to another block. The predicate evaluation method for different blocks

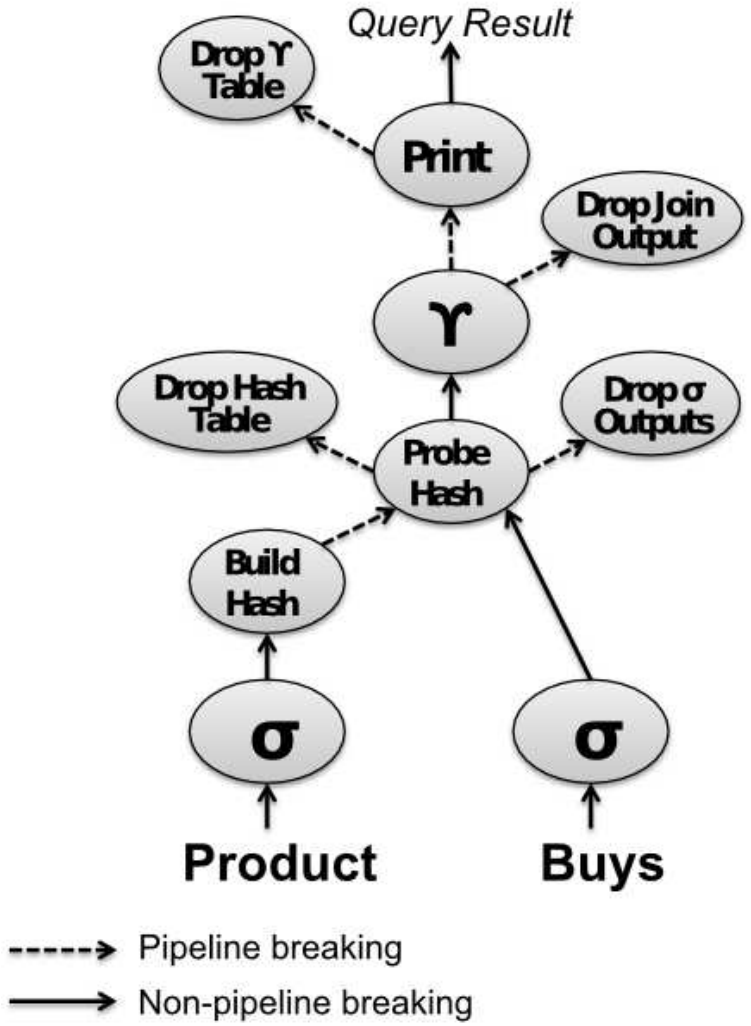


Figure 2.2: DAG plan for the sample query.

may be different, subject to their layout, availability of indexes inside the block etc. The complexity of predicate evaluation is abstracted within a block (cf. Section 2.3), and the worker thread and work order code need not know anything about it.

Note that in the example query, the hash table is built on the output of the select operator on the Product table. The edge connecting the select operator and the build hash operator allows data pipelining. As soon as a filled block of output from an upstream operator (the select operator in this case) is available, it is streamed to its consumer operator (the build hash operator). As soon as some input is available, a work order is created which can then be scheduled by the foreman. Scheduling a work order is accomplished by simply sending a message to a particular worker over the TMB. The scheduler policy can

be tuned to favor different execution models, such as aiming of high performance, staying with a certain level of concurrency/thread-level parallelism/CPU resource consumption for a query, etc. Currently, there is only an eager implementation that schedules work orders as soon as they are available, with execution of work orders for a given worker proceeding in FIFO order (as new work orders are generated, the foreman attempts some load balancing by assigning new work to workers that have little or no pending work). There is significant research in designing appropriate scheduling policies, which is ongoing work that has been undertaken by other members of the Quickstep development team. The key point to note is that by separating the scheduling policy from the scheduling mechanism the design allows for easily varying the scheduling policy, and hence the overall system behavior.

To begin the probe phase of the hash join, the building of the hash table needs to be complete (note the pipeline-breaking dependency between the probe and the build operator in the DAG). After the build operator has completed, the foreman is free to schedule a work order for each full block of tuples that is produced by the select operator that is working on the Buys table. (Quickstep has a mechanism where the work orders signal to the scheduler that a block is full. This mechanism ensures that if a single selection work order produces a partially filled block, then a handle to that partially filled block is available to another selection work order that runs later. Thus, there is minimal internal fragmentation for blocks as query execution moves through the DAG.)

The probe hash work order execution involves checking the hash table for match(es) and writing them to temporary output block(s). The edge from the probe hash operator to the aggregate operator allows for data pipelining. Thus, fully-filled output blocks from the probe hash operator can be streamed to the aggregation operator (shown using the symbol γ in the figure).

Finally, note the various drop operators in the DAG shown in Figure 2.2. These operators are used to drop the "temporary" data that is materialized during query execution and is no longer needed beyond a certain stage of the DAG. Recall the structures such as hash tables are created in buffer pool memory, so all memory management is purposely designed to be within the purview of the buffer manager.

Work Order Execution

The work orders are implemented as C++ classes, and there is one implementation for each operator in the system. Each work order class implements a virtual, thread-safe execute()

method which contains the implementation of the work order. Each worker thread simply executes this method. This design make it easy to add new operators or to extend an existing operator.

Partitioning

Quickstep also has the notion of partitioning a table, which can be used to control the distribution of data in a table in NUMA machines.

A relation can be partitioned horizontally using two types of partitioning: Hash and Range. Both base and temporary relations can be partitioned (across the NUMA memory). Every relation that is partitioned has a *partition scheme* associated with it. A partition scheme contains the details about the type of partitioning (hash or range), which attributes' values are used for partitioning, the number of partitions, and which blocks of the relation belong to which partitions.

When tuples are inserted into a relation, the partition scheme determines which partition each tuple belongs to based on its own internal policy (e.g. computing the hash of a particular attribute's value and then computing the remainder modulo the number of partitions), and divides the tuples so that they are inserted into blocks belonging to the appropriate partition.

The following example illustrates hash partitioning, where the hash function for integers is the identity function. Consider a relation R, with a single attribute of type integer, which is also the partitioning attribute. The relation is hash partitioned with 4 partitions. In this scenario, when a tuple with value 0 or 4 is inserted, the tuple is placed in a block belonging to partition 0. When the next tuple with value 1 or 5 is inserted, although there is space left in the block that was previously created, a new block is created since this tuple belongs to partition 1. In a similar way, all the tuples belonging to a particular partition are inserted into blocks that belong to that partition.

Whenever an operator in Quickstep needs to write data to blocks, it uses a class called *InsertDestination*, which has methods for inserting tuples one-at-a-time or in bulk. The InsertDestination interface is responsible for the common tasks of pushing data into blocks, creating new empty blocks as needed, and signaling to the foreman when blocks are full and ready to be consumed. InsertDestinations have an associated partition scheme (the default is no partitioning). When inserting tuples into an InsertDestination, a pass is first made to assign each tuple to a partition (querying the Partition Scheme), then tuples are bulk-inserted into blocks belonging to the appropriate partition, with new blocks being

created in each partition when needed. By implementing partitioning as an option in the common tuple-insertion path, relations can easily be partitioned when data is loaded *or* the output of any operator can be repartitioned on-the-fly as part of normal query execution.

To illustrate how partitioning works, consider the following query:

```
SELECT L_ORDERKEY
FROM LineItem NATURAL JOIN Orders;
```

Assume that the relations LineItem and Orders are partitioned into four partitions each, using hash-partitioning on the join key when they are created. When the data is loaded for these relations, each tuple is inserted into a block belonging to the appropriate partition. If the system has four sockets, then each corresponding partition of both the relations are placed across the four sockets – one partition-pair per socket. When computing the hash join between these two relations, four hash tables are created – one per partition-pair across the four (NUMA) sockets. Work orders for both the build and the probe phase of the hash join will access local memory on only one NUMA socket for input blocks, the hash table, and for materializing join output.

The scheduler (described above) is also NUMA-aware, and aims to send each work order to a worker thread running on the NUMA node where the input blocks referred to in the work order reside. The scheduler can be configured to be *strict* with regard to NUMA-aware scheduling (only giving workers NUMA-local work orders), or to allow a relaxed *work-stealing* policy, where idle workers on other sockets can be assigned non-local work orders at times when there are no NUMA-local work orders for them to process.

WideTables

Quickstep can also use a schema-based denormalization technique called WideTable [71]. This technique walks through a schema graph and converts all foreign-key primary-key "links" to an outer-join expression (to preserve NULL semantics). The resulting flattened-out table is called a WideTable, and is essentially a denormalized view of the entire database. The columns in this WideTable are stored as column-stores, and BitWeaving indices can be additionally built on columns of interest.

This type of denormalization is largely agnostic of workload characteristics (it is a schema-based transformation). It has been acknowledged that such techniques are expensive for databases that are updated often, and the implementation in Quickstep is rudi-

mentary – there is currently no way to recreate the WideTable on updates. Techniques for incrementally updating the WideTable (rather than recreating it from scratch) when new data is appended to existing tables can be envisioned, but they have not yet been implemented.

In addition, Quickstep has a relatively simple rule-based optimizer implementation that currently only allows for simple transformation of queries. As a result, only a limited class of queries can be answered for databases on which WideTables are built. Note that Quickstep does allow running queries using regular joins, so when a WideTable is dropped and has to be reconstructed, the system can still execute queries using traditional methods (although the performance may be lower).

2.5 Related Work

The focus of this chapter is on high performance in-memory analytics computing, and this chapter narrows its focus to single machine NUMA settings. There is tremendous interest in the broader overall main-memory database area, including MonetDB [25, 26, 53], Blink [16, 85], Hyper [60], Shark [110]), SparkSQL [11], Vectorwise [113], SAP HANA [36], and IBM DB2 BLU [84]). This work has similar inspiration as these other projects.

Within the area of fast columnar scans, there have been a number of recent proposals [1, 57, 70, 83, 85, 105, 106, 112], and BitWeaving is largely used as an index in Quickstep, which falls under a broader class of efficient bit-based indexing/storage methods that are optimized for modern processors [16, 29, 37, 38, 57, 58, 66, 79, 85, 89, 104, 108, 109]. CSB+Tree [87] is also used as another indexing structure.

The implications of NUMA architectures for performance is well-known and has been a subject of significant research over the past few years. Some of the earliest work on query processing for NUMA includes the work by Bouganim et al. [27]. As NUMA architectures have become more main-stream, there has been a renewed interest in determining the impact of such change on analytical query processing [12, 14, 17, 23, 24, 61, 62, 69].

The use of a unique block-based storage architecture (which is explored in more detail in Chapter 3) naturally leads to a block-based scheduling method for query processing. The recent morsel-based query processing [68] method also philosophically belongs to this style of query processing, and in this work we build on these ideas.

Overall, the key contribution of this chapter is presenting a holistic system that employs a unique and flexible scheduler-based query processing architecture, with novel indexing

and data normalization methods that are optimized for NUMA settings for in-memory analytical data processing environments.

2.6 Quickstep Development & Acknowledgments

It should be noted that the development of Quickstep as described has been a very collaborative process, and the features described in this chapter were developed together with other students, in particular Yinan Li, Harshad Deshmukh, Qiang Zeng, Shoban Chandrabose, Zuyu Zhang, Adalbert Gerald Soosai Raj, James Paton, and Sangmin Shin.

BitWeaving and WideTable were both conceived of and implemented by Yinan Li, and are discussed in much greater detail in his dissertation *Analytic Query Processing at Bare Metal Speeds*. Future publications and theses from other students will similarly expand on aspects of Quickstep that are primarily their work, in particular the scheduler (Harshad Deshmukh), partitioning and NUMA-awareness (Adalbert Gerald Soosai Raj), sorting and aggregation features (Shoban Chandrabose), and the development of a distributed version of Quickstep (Zuyu Zhang).

3 STORAGE ORGANIZATIONS FOR READ-OPTIMIZED MAIN MEMORY DATABASES

3.1 Introduction

Dropping DRAM prices and increasing memory densities have now made it possible to economically build high performance analytics systems that keep their data in main memory all (or nearly all) the time. There is now a surge of interest in main memory database management systems (DBMSs), with various research and commercial projects that target this setting [25, 33, 35, 46, 60, 64, 75, 85, 103, 113].

This chapter presents results from an experimental survey/evaluation of various storage organization techniques for a main memory analytical data processing engine, i.e. the focus on a read-optimized database setting. Note that the focus of this chapter is on *experimental evaluation* of a number of existing storage organization techniques that have been used in a variety of DBMS settings (many in traditional disk-based settings), but the central contribution of this chapter is to investigate these techniques for main memory analytic data processing. In addition, an experimental platform has been made publicly available for the community to use to design and evaluate other storage organization techniques for read-optimized main memory databases.

Designers of read-optimized main memory database storage engines have to make a number of design decisions for the data storage organization. Some systems use column-stores for the data representation, whereas others use row-stores. The use of indexing is not generally well understood or characterized in this setting. Other questions such as to whether it is advantageous to store the data for a given relation in large (main memory) "files" or segments, or to break up the data into traditional pages (as is done for disk-based systems), or to choose some intermediate data-partitioning design, are also not well characterized. In this chapter, an experimental approach is used to consider and evaluate various storage-related design tradeoffs.

The approach taken for this research is to build a prototype main memory storage engine with a flexible architecture, which then allows for a study of the cross section of storage organization designs that is produced by examining the following dimensions: a) large memory "files" vs. self-contained memory blocks, b) row-store vs. column-store, c) indexing vs. no indexing, d) compression vs. no compression. This chapter describes a comprehensive study of the storage design space along these axes and a number of in-

teresting experimental findings. For example, experiments showed that a novel block-based storage organization outperforms file-based organization at both query time and load time, and that cache-sensitive index structures have a major role to play in accelerating query performance in this main memory setting.

Overall, the key contributions of this chapter are as follows: First, there is a systematic characterization of the design space for main memory database physical storage organizations using the four dimensions listed above.

Second, a flexible storage manager design that facilitates comparing the different storage organization choices is presented. This storage manager has an interesting block-based design that allows for a flexible internal organization. This framework allows direct comparison of a number of storage alternatives, and it may serve as a platform for other researchers to improve on the methods described here and perhaps design and evaluate other storage organization techniques.

Third, a rigorous, comprehensive experimental evaluation was conducted to characterize the performance of selection-based analytic queries across the design space, providing a complete view of the various storage organization alternatives for main-memory data processing engines. The study reveals several interesting experimental findings, including that both row-stores and column-stores have performance sweet spots in this setting, and can coexist in the Quickstep storage manager design. Experimental results also show that indexing and compression continue to play an important role for main memory storage, and are often required for high performance.

Finally, the results in this chapter allow designers of main memory storage engines to make informed decisions about tradeoffs associated with different storage organizations, such as figuring out what they leave on the table if they use a pure column-store vs. supporting both row-stores and column-stores.

Note that, to keep this work focused on the core storage management aspect (as opposed to query evaluation algorithms, query optimization, etc.), the evaluation in this chapter largely covers single relational access plans, which form the building blocks for more complex query evaluation methods. Furthermore, previous work on main memory analytical databases has emphasized the importance of these simple scan-based access methods [16, 35, 92]. This approach keeps experiments focused on key storage organization issues. More complex query processing mechanisms can interact with the storage engine in interesting ways, but they still need fast support for basic selection operations, and currently there isn't a clear consensus on how to best build a main memory query

processing engine; for example, even single join methods are being rethought in this setting [5, 13, 23, 63].

The remainder of this chapter is organized as follows: Section 3.2 describes the Quickstep Storage Manager, a platform for read-optimized main memory storage experiments. Section 3.3 describes the experimental setup that is used in this chapter, and Section 4.5 presents the experimental results. Related work is discussed in Section 3.6, and Section 3.7 contains concluding remarks.

3.2 Quickstep Storage Manager

The Quickstep Storage Manager (SM) is used as the experimental platform in this chapter to systematically evaluate storage organizations for read-optimized main memory DBMSs. This Storage Manager has a flexible architecture that naturally allows direct comparisons of various storage organization choices within the same framework.

Designing a new storage engine with support for many different physical data organizations allows for experiments to be conducted that would not otherwise have been possible. For instance, it would be difficult to draw any meaningful conclusions about the relative performance of row-store and column-store layouts by simply comparing the performance of a main-memory DBMS that uses row-stores (e.g. VoltDB [103]) with one that uses column-stores (e.g. VectorWise [113]), since the entire code base and execution model is different. It should be noted that some DBMSes like MySQL [80] allow some flexibility in terms of storage within the same database by providing a "plug-in" system for different storage engines. Nevertheless, there are some assumptions about the access patterns for data that are built-in to the interfaces that such plug-in storage engines must implement (for instance, the MySQL storage engine interface is oriented towards row-at-a-time "cursor" access, which would prevent many of the possible benefits of a column-store from being realized).

The Quickstep storage manager allows for different design choices in several dimensions of in-memory organization to be evaluated directly against each other, and it has a high-level, logical relational API that does not prejudice it towards any particular pattern of data access.

The Quickstep SM Architecture

The basic unit of organization in the Quickstep SM is a *storage block*. A table is typically stored across many blocks, though each block belongs to one and only one table.

From the perspective of the rest of the system, blocks are opaque units of storage that support a select operation. A select operation is specified as a projection list, a predicate, and a destination. The destination is some (temporary) block(s) where the results should be materialized. Blocks are independent and self-contained, and each one can decide for itself, based on its own internal organization, how to most efficiently evaluate the predicate and perform the selection. Blocks support update and delete operations that are also described logically and performed entirely within the storage system.

Like System R [91], the Quickstep SM allows pushing down single-table predicates to the SM, but Quickstep goes beyond System R in that Quickstep's expression and predicate framework allows *any* arbitrary predicate to be evaluated entirely inside the storage system, so long as it only references attributes of a single table (i.e. it is not limited to a narrow definition of "sargable" predicates). Like MonetDB [25] and Vectorwise [113], the Quickstep SM does away with the traditional tuple-at-a-time cursor interface, and materializes the complete result of a selection-projection operation on a block all at once into other in-memory block(s).

The Quickstep SM organizes the main memory into a large pool of memory "slots" that are occupied by storage blocks. A given table's tuples are stored in a number of blocks, and blocks may be different sizes (modulo the size of a slot), and have different physical layouts internally. The storage manager is responsible for creating and deleting blocks, and staging blocks to persistent storage (SSD, disk, etc.).

Internally, a storage block consists of a single *tuple-storage sub-block* and any number of *index sub-blocks*. The index sub-blocks contain index-related data for tuples in that block and are are self-contained in the storage block. The index sub-blocks can be viewed as partitioned indices [43].

The block-oriented design of Quickstep's storage system offers a tremendous amount of flexibility with regards to physical database organization. Many different tuple-storage sub-block and index sub-block implementations are possible and can be combined freely (the particular sub-block types studied in detail are described in Section 3.3). Not only is it possible for different tables to have different physical layouts, it is possible for different blocks *within* a table to have different layouts. For instance, the bulk of a table could be stored in a sorted column-store format which is amenable to large-scale analytic queries,

while a small "hot" group of indexed row-store blocks can support a live transactional workload.

Storage blocks are a natural unit of parallelism for multithreaded query execution, and temporary blocks are units of data flow between operators in complex queries.

3.3 Experiments

As discussed in the Introduction, the focus of this study is on core storage manager performance for read-optimized main memory settings in which selection-based queries are common. The workload builds on a previous study [51] that had a similar goal, but in a disk-based setting. Specifically, the experiments use the tables and queries that are described below.

Tables

The following set of four tables are used in experiments:

- **Narrow-U:** This table has ten 32-bit integer columns and 750 million rows. The values for each column are randomly generated in the range [1 to 100,000,000].
- **Narrow-E:** This table has ten 32-bit integer columns and 750 million rows. The values for column i are randomly generated in the range from [1 to $2^{2.7*i}$].
- **Wide-E:** A table similar to Narrow-E, except that it has 50 columns instead of 10 and 150 million rows instead of 750 million. Thus, the total size of the generated data is the same. The values for column i are randomly generated in the range from [1 to $2^{4+(23/50)*i}$].
- **Strings:** This table has ten 20-byte string columns and 150 million rows. Each string is a random sequence of mixed-case letters, digits, spaces, and periods.

As in the previous disk-based study [51], it is primarily integer columns that are studied. Integer types are common and generalize nicely. Some experiments also consider a table composed entirely of strings, as their storage requires more space, and they are a commonly encountered data type.

Queries

The workload primarily consists of queries of the form:

```
SELECT COL_A,COL_B,... FROM TBL WHERE COL_A >= X;
```

The projected columns `COL_A ...` are randomly chosen for each experiment. Each query has a predicate on a single column `COL_A`. The literal value `X` is chosen based on the range of values for `COL_A` to achieve a desired selectivity. These experiments consider predicates with selectivity 0.1%, 1%, 10%, 50%, and 100%.

The number of columns that are projected is also varied. For the Narrow-U, Narrow-E, and Strings tables, which each have ten columns, 1, 3, 5, or 10 columns are projected in different experiments. For Wide-E, which contains 50 columns, 1, 15, 25, or 50 columns are projected. For each query, a random subset of the desired number of columns is chosen to project. For each table, there are also experiments which only measure the time to evaluate a predicate without projecting any values or materializing any output (this is reported as a projection of zero attributes).

With the exception of zero-projection queries (which produce no output), the output of each query is materialized in-memory in a simple non-indexed row-store format.

Experiments with more complex predicates are also conducted to determine whether observations for simple predicates hold. As in the previous study [51], the complex predicates that are studied are conjunctions of three single-column predicates ANDed together. As in other experiments, the selectivity and number of columns projected is varied.

Finally, some experiments with an aggregate query are conducted to gain some insight into how the storage organizations studied here affect the performance of higher-level query-processing operations. The experimental aggregate query is directly based on the aggregate queries Q21 and Q24 (`MIN` with 100 partitions) from the Wisconsin Benchmark [32], and has the following form:

```
SELECT ONEPERCENT, MIN(COL_A) FROM TBL GROUP BY ONEPERCENT HAVING MIN(COL_A) < X;
```

The Narrow-U table's schema is slightly modified for the aggregate experiment, replacing one of the standard columns with a `ONEPERCENT` column which has 100 unique values and is used as the group-by attribute.

As in previous work [2, 49, 51], the performance metric used for evaluation is single query response time, which has a large impact on overall system performance. Individual query response time is also an important metric in interactive analysis environments.

Physical Database Organization

This chapter explores a number of key dimensions related to physical organizations for read-optimized main memory DBMSs. These dimension are described below.

Files vs. Blocks

In traditional DBMSs, tuples are stored in large files which are subdivided into pages (disk or memory pages). Pages are the units of buffer management and disk I/O. Non-clustered indices are separate files, external to the base table files which they reference. Modern column-store DBMSs store projections of distinct columns as separate column-striped files, which nonetheless fit into the large-file paradigm [65, 96]. Even where modern main memory DBMSs have abandoned page-based buffer management and I/O, they typically still organize data into large file-like memory segments.

As described in Section 3.2, Quickstep's SM is built around the concept of "blocks" as self-contained, self-describing units of storage. A single table's tuples are horizontally partitioned across many blocks. Internally, blocks have a tuple-storage sub-block which stores complete tuples, and any number of index sub-blocks that index the tuples in the tuple-storage sub-block. Multiple implementations of both types of sub-block, with different physical layouts, are possible. The choice of sub-blocks is represented by additional dimensions in experiments below.

The first major dimension of experiments is a comparison between a traditional large-file layout and a block-oriented layout where tuples are divided amongst blocks and indices, if any, are co-located with data inside blocks.

Block Size & Parallelism An important consideration for the block-oriented layout is the size of blocks (Quickstep allows differently-sized blocks). A sub-experiment was conducted where the size of blocks was varied by powers of 2 from 128 KB to 256 MB and the response times of queries of the form described in Section 3.3 on the Narrow-U table were recorded. The number of worker threads used to process queries in this experiment was also varied, using 1, 2, 5, or 10 threads pinned to individual CPU cores, and 20 threads with one thread pinned to each hardware thread of a 10-core hyperthreading-enabled CPU. The results of this experiment are reported in detail in Section 3.4. In summary, a block size of 16 MB with 20 hyperthreading-enabled worker threads resulted in good performance across the mix of queries and other physical organization parameters that were tested, so the number of worker threads was fixed at 20 and the block size was fixed at 16 MB for other experiments.

Partitioning The block-oriented design also allows tuples to be partitioned into different groups of blocks within a table depending on the value of some column(s). A number of strategies for assigning tuples to partitions is possible, analogous to the choices for horizontal partitioning in a clustered parallel database system [73, 88]. For the queries considered here, which involve a range-based predicate on a single column, experiments were conducted with range-based partitioning based on the value of a single column. These experiments use 16 partitions evenly divided by value range [1]. For all but the 100% selectivity queries, this has the effect of assigning all relevant tuples for a predicate on the partition-column to a limited subset of blocks.

Row-Stores vs. Column-Stores

The relative benefits of row-store and column-store organization for read-optimized databases in disk-based settings have been extensively studied [2, 49, 51]. This chapter considers the impact of this design choice in main memory settings. Both a conventional unsorted row-store and a column-store sorted on a single primary column are evaluated in experiments. Both layouts are evaluated in a large-file and block-oriented context, with and without secondary indices.

Secondary Indices

Secondary indices can often speed evaluation of predicates when compared to scans of a base table. A cache-sensitive B+-tree [87] index is implemented in the Quickstep SM with a node size of 64 bytes (equal to the cache-line size of the test system). The response time of test queries is measured when evaluating predicates via a CSB+-tree on the appropriate column's values for both row-store and column-store layouts, with both the large-file and block-oriented designs. The response time using the index is compared to predicate evaluation using scans and a binary search of the sorted column for the column-store organization.

Compression

Compression is often effective in disk-based read-optimized database settings [51]. An important consideration for main memory read-optimized databases is whether the ad-

[1] The choice of 16 partitions is somewhat arbitrary, and is meant to illustrate the potential benefits of partitioning. A full study of partitioning strategies (including handling of skew) is beyond the scope of this chapter.

vantages of compression still apply for data that is entirely memory-resident. Dictionary-coding and bit-packing[2] compression techniques are implemented in the Quickstep storage manager. Compression is available for both row-stores and column-stores, and if a CSB+-Tree index is built on a compressed column, it will also store compressed codes. All queries considered in experiments can work directly on compressed codes. The Quickstep compression implementation builds up sorted dictionaries for each column as a block is being built and, if space can be saved by compressing that column, stores compressed codes (which still compare in the same order) instead of native values. Dictionaries for compressed columns are stored inside blocks. For integer columns whose values are in a limited range, bit-packing is applied *without* a compression dictionary by simply truncating these values down to their lower-order bits.

The difference in performance by using compression is measured in block-oriented organization on the Narrow-E table, which contains several compressible columns. Various experiments combine compression with both row-store and column-store layouts, with and without indexing.

Experimental Setup

Experiments were run on a four-processor Intel Xeon E7-4850 server with 256 GB of SDRAM in a NUMA configuration running Scientific Linux 6. Each processor has 10 cores and is clocked at 2.0 GHz. Each core has dedicated 32 KB L1 instruction and data caches and a dedicated 256 KB L2 cache, while each processor has a shared 24 MB L3 cache. The cache-line size is 64 bytes. Because the experiments in this chapter are focused on the performance impact of data organization, most experiments are run on a single CPU socket (with up to 20 threads) using locally attached memory (64 GB). Some supplemental scale-up experiments that use all four CPU sockets are also conducted.

The experiments are run by a test-driver executable which generates a table from Section 3.3 in memory with a specified storage organization, and then proceeds to run a series of experiments, varying query selectivity and projection width as described in Section 3.3. For each combination of physical organization, table, and query parameters, 10 query runs are performed and the total response time for each is measured. The mean and standard deviation of the execution times are reported. The total number of L2 and L3 cache misses incurred during each query run are also measured using hardware per-

[2]The bit-packing implementation in Quickstep pads individual codes to 1, 2, or 4 bytes, as it was found that the cost to reconstruct codes that span across more than one word significantly diminished performance.

formance counters on the CPU. For these experiments, Quickstep is compiled with GCC 4.8.0 using `-march=native` and optimization level `-O3`.

In the initial block-size experiments, it was found that using 20 worker threads with hyperthreading enabled tended to produce the best performance, so all subsequent queries are run using 20 worker threads. Queries are run one-at-a-time, with individual worker threads operating on different blocks for intra-query parallelism. Since the focus of this chapter is on read-optimized databases, no sophisticated transactional concurrency control or recovery mechanisms are enabled (queries simply grab table-level locks).

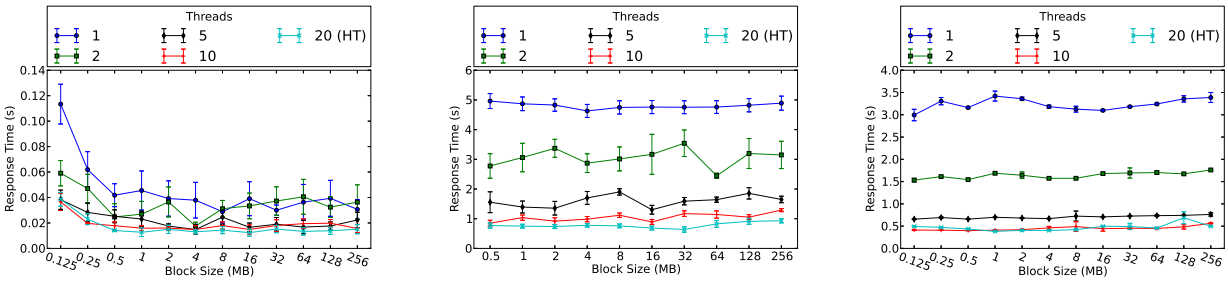
3.4 Results

Most of the results reported in this section are based on queries on the Narrow-U table. How these findings are affected by the presence of wide attributes (as in the Strings table) and wide rows (as in the Wide-E table) are discussed later in this section.

From Figure 3.2 onward, a series of graphs is presented that compares the performance of various options in the large space of storage organizations studied in these experiments. All graphs show total query response time on the vertical axis (lower is always better) and vary either the selectivity or the projection width of test queries on the horizontal axis. Error bars show the variation in query response time across 10 experiment runs. To make these graphs easier to read, each graph is accompanied by a table that identifies what region of the design space is being shown (the organization, tuple layout, and indexing), with the dimension that is being examined shown in **bold text**. The selectivity and projection width of queries are also indicated (for any given graph, one will be fixed and the other varied along the horizontal axis).

Block Size & Threading Experiment

An experiment was conducted to determine how to tune the size of blocks in the block-oriented layout, and the number of worker threads used for intra-query parallel processing. For this experiment, the entire set of queries described in Section 3.3 were run on a smaller version of the Narrow-U table consisting of 100 million tuples. The size of blocks was varied by powers of two from 128 KB to 256 MB, and the number of threads from 1 to 20 (in the 1-10 thread cases, the threads are pinned to individual CPU cores, while with 20 threads, the threads are pinned to individual hardware threads with 2 threads



(a) 0.1% selectivity, 3 attributes projected, predicate on sort column of Column Store.

(b) 10% selectivity, 3 attributes projected, Row Store with CSB+-Tree index.

(c) 1% selectivity, 3 attributes projected, Row Store (scan).

Figure 3.1: Block size vs. response time.

per hyperthreading-enabled core). Three queries are presented below that represent the key behavior that was observed across the entire set of queries that were run.

Figure 3.1a shows a query with a 0.1% selectivity predicate on a column store's sort column. Observe that optimal block sizes are in the range of 16-32 MB, and that adding threads for intra-query parallelism results in substantial speedup. Also observe that hyperthreading is useful in speeding up query execution, because predicate evaluation in this layout involves a binary search of the sorted column, a random-access pattern of memory accesses which is not well-handled by prefetching. When one thread incurs a cache miss, it is often possible for the other thread on the same core to immediately take over and do some work, effectively "hiding" the cost of many cache misses. In the example shown, at 16 MB block size, enabling hyperthreading reduces response time by 17.3% over the 10 thread case, even though the average total number of L3 cache misses is similar (35790 without hyperthreading and 37202 with hyperthreading). The 16-32 MB range of block sizes is well-tuned to allow individual worker threads to often hit cache lines that had already been faulted or prefetched as part of previous "random" accesses.

Next, Figure 3.1b shows the result for a 10% selectivity query using a row store with a CSB+-Tree index. Similar to the column store case above, the optimal block size is approximately 16-32 MB. Again, adding threads for intra-query parallelism improves the query performance. Hyperthreading again improves performance because it is able to "hide" the cost of cache misses arising from random access (using a non-clustered index produces a sequence of matching tuples in an order other than their physical order in the base row-store, so tuples are accessed in a random order when the projection is performed). In the example shown, at 16 MB, hyperthreading reduces overall response time by 23.1%,

even though the number of L3 cache misses incurred is actually slightly greater (18.4 million without hyperthreading and 19.0 million with hyperthreading). Similar results were observed when using a CSB+-Tree to evaluate a predicate on a non-sorted column of a column store.

Observation 1. *A block size of 16-32 MB with hyperthreading enabled and all hardware threads in use provides optimal or near-optimal performance for combinations of storage formats and queries that involve random access.*

The last query for this experiment is a simple scan query. This result is shown in Figure 3.1c. Scan-based queries have a linear access pattern which works well with cache prefetching. Hence, there is little variation in response time for block sizes below 64 MB (most cache misses are avoided by prefetching), and cache misses are infrequent enough that hyperthreading does not significantly improve performance. The results are similar when scanning a column store.

Observation 2. *Scan-based queries are less sensitive to block size and perform well for all block sizes below 64 MB. Hyperthreading is not beneficial (but also not harmful) for scan-based queries, and query performance increases with additional threads up to the number of physical CPU cores.*

Based on these observations, for the rest of the experiments, the block size is fixed at 16 MB and the number of worker threads is fixed at 20.

Relationship to Cache Size & Number of Cores

Optimal block size is related both to the CPU's cache size and its number of cores (since multiple cores share a unified L3 cache). To test this, block-size experiments were also conducted on a machine with a Core i7-3610QM CPU (4 cores/8 threads with 6 MB of L3 cache), and one with a Xeon X5650 CPU (6 cores/12 threads with 12 MB of L3 cache). The best-performing block size is in the range of 8-16 MB for the Core i7 machine, and about 16 MB for the Xeon X5650 machine. This suggests a rule of thumb for tuning block size: the optimal block size is approximately 5X to 10X times the L3 cache size divided by the number of cores [3]. Cache locality is important for performance, but the optimal block size somewhat "overuses" the L3 cache, because prefetching (especially when scanning)

[3]This is a purely empirical observation based on performance measurements on these three Intel CPUs. On systems with significantly different hardware (particularly with respect to memory bandwidth or cache hierarchy and capacity), this "rule" may not apply, and benchmarks should be repeated to appropriately tune the block size.

can reduce the number of cache misses, and hyperthreading can mitigate the performance impact of cache misses when they occur.

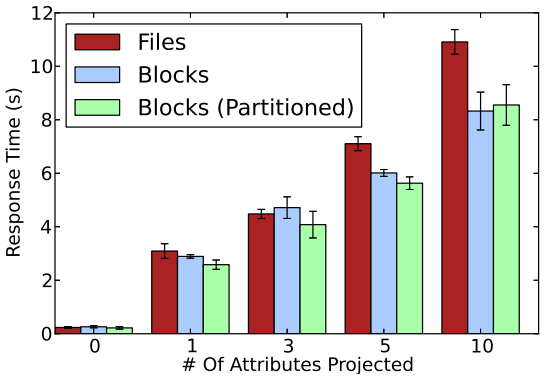
Multi-Socket NUMA Scale-Up

Queries similar to those illustrated in Figures 3.1a-3.1c were run with a larger 3 billion row dataset using all four sockets in the multi-socket NUMA server. Increasing the number of cores used on a single socket yielded good, near-linear speedup (10 cores were 6.5X-9X as fast as one core, depending on the query). However, using 20 cores on 2 sockets was never more than 20% faster than using 10 cores on a single socket, and going to 3 or 4 sockets actually caused response time to increase. The poor NUMA scaling observed in these experiments is a consequence of the fact that the experimental code-base used in this chapter is NUMA-oblivious, and neither the creation of blocks nor the assignment of blocks to worker threads is done with awareness of the different access costs for memory attached to different sockets or the contention for bandwidth on inter-socket links, causing worker threads to frequently stall waiting for data from non-local memory. The results here confirm the need for NUMA-aware data placement and processing observed in previous research [5, 107], which is an active area of research in the community, and an important area of future research with Quickstep.

Files vs. Blocks

The traditional large-file organization was compared with Quickstep's block-oriented organization across the other dimensions of the experiments. So as not to penalize the file-based organization for a lack of parallelism, the test tables are statically partitioned into 20 equally-sized files for 20 worker threads to operate on.

Figure 3.2 shows the difference in performance for queries at 10% selectivity where the predicate is on the sorted column of a column-store (for partitioned blocks, this is also the column whose value the blocks are partitioned on). The performance of files is similar to the performance of blocks for narrow projections. When projecting 5 or 10 columns, blocks outperform files due to improved locality of access and caching behavior when accessing several column stripes within a relatively small block to reassemble tuples (in the example shown, file-based organization incurs 77.4 million total L3 cache misses and 1.53 billion L2 misses, while block-based organization incurs only 73.6 million L3 misses and 1.14 billion

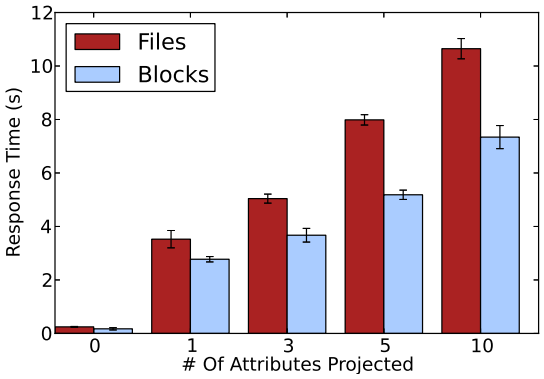


Organization	File vs. Blocks
Tuple Storage Layout	Column-Store
Indexing	None
Predicate	10% Selectivity On Sorted Column
Projection Width	Varies

Figure 3.2: Files vs. Blocks – Sorted Column

L2 misses). This same pattern of results occurs at the other selectivity factors that were tested.

Observation 3. *For queries with a predicate on the sorted column of a column-store, using a file or a block organization makes little difference in performance, except for wide projections, where blocks tend to perform better.*



Organization	File vs. Blocks
Tuple Storage Layout	Row-Store
Indexing	CSB+-Tree
Predicate	10% Selectivity On Indexed Column
Projection Width	Varies

Figure 3.3: Files vs. Blocks – Non-Sorted Column

Figure 3.3 shows the difference in performance for queries at 10% selectivity where the predicate is on a non-sorted column of a row-store, and a CSB+-Tree index is used for predicate evaluation. Across all projection widths, the block-oriented organization outperforms the file-based organization thanks to improved locality of access in relatively small blocks (in the example shown, when projecting 5 attributes, file-based organization incurred 166 million L3 cache misses and 1.63 billion L2 misses, while block-based organization incurred 177 million L3 misses but only 673 million L2 misses). This result holds

across all of the selectivity factors tested. For projections of one or more attributes, reduction in response time for blocks vs. files ranges from 2.1% (projecting 1 attribute at 1% selectivity) to 37.7% (projecting 10 attributes at 50% selectivity). Similar reductions in response times for blocks vs. files were observed when evaluating a predicate on an unsorted column of a column-store with a CSB+-Tree index, and when evaluating predicates via a scan on a row-store or column-store.

Observation 4. *For queries with a predicate on a non-sorted column, a block-based organization outperforms a file-based organization. This result holds for both row stores and column stores, with and without indexing.*

The block-based organization in Quickstep adds little storage overhead compared to files (there is less than 100 bytes of additional metadata per block). The total memory footprint of the Narrow-U table without indices is 28610 MB for files and 28624 MB for blocks (0.05% additional storage for blocks). With a secondary CSB+-Tree index, the total memory footprint is 35763 MB for files and 35776 MB for blocks (0.04% additional storage for blocks).

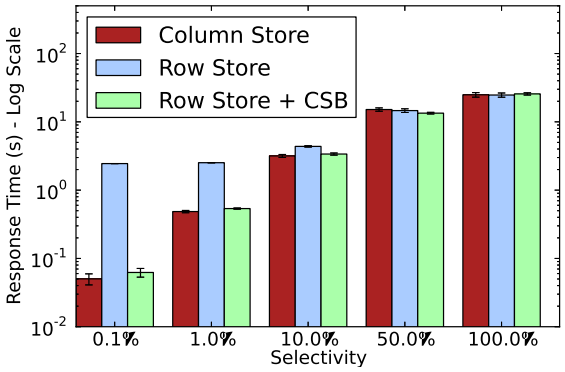
Column-Store Load Cost

It should be noted that the cost of building large sorted column-store files is higher than the cost of building many individual sorted column-store blocks. In these experiments, it takes 159.8 seconds to sort the 750 million tuple Narrow-U table into 20 partitioned column-store files (using 20 worker threads, 1 per partition). Building sorted 16 MB column-store blocks, on the other hand, takes only 87.8 seconds (only 55% as much time, again using 20 worker threads). The sort algorithm used in both cases is introsort [76]. As seen in Figure 3.2, the read-performance advantages of a sorted column-store are maintained in the block-based layout, but the initial load time is smaller.

Observation 5. *The build time for a sorted column-store in a block-based organization is smaller than that for a file-based organization, but gives the same or better read query performance.*

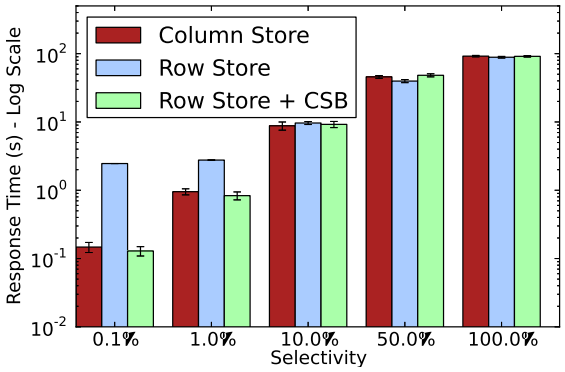
Row-Store vs. Column-Store

Comparing row-store and column-store tuple storage layouts in blocks, it was found that, when selecting via a predicate on the sorted column of the column-store, the column-store outperforms the row-store for narrow projections, whether or not a secondary CSB+-



Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None/CSB+-Tree
Predicate	On Sorted/Indexed Column
Projection Width	1 Column

Figure 3.4: Column-Store vs. Row-Store (+Index) – Narrow Projection

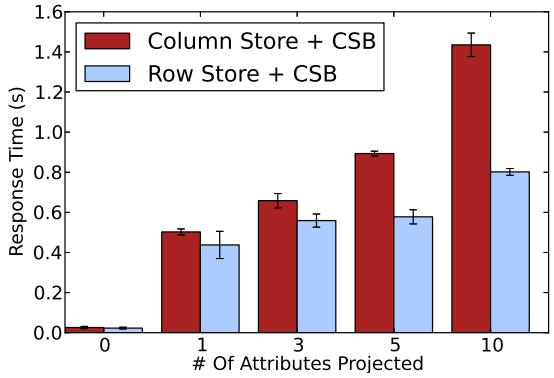


Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None vs. CSB+-Tree
Predicate	On Sorted/Indexed Column
Projection Width	10 Columns

Figure 3.5: Column-Store vs. Row-Store (+Index) – Wide Projection

Tree index was built on the row-store to speed predicate evaluation (indexing is discussed further below). As one might expect, predicate evaluation is fast for the column-store, merely requiring a binary search on the sorted column. After this search, performing the actual projection involves linear access to a contiguous region of a few column stripes. The column-store's performance advantage for narrow projections is illustrated in Figure 3.4. For wider projections (5 or 10 attributes), the performance of the column-store is nearly equal to that of the row-store with a secondary index, as seen in Figure 3.5. Although predicate evaluation is more complicated when using an index, and results in a random access pattern for tuples, the row store makes up for this by storing all the column values needed for a projection on one tuple in one or two contiguous cache lines instead of several disjoint column stripes. Similar results were observed when comparing column-stores and row-stores in the large-file organization (in fact, the performance advantage for column-stores doing narrow projections was more pronounced).

Observation 6. *For queries with a predicate on a column-store's sorted column, a column-store outperforms a row-store (even when using an index on the appropriate column of the row-store) for narrow projections. For wider projections, performance of column-stores and row-stores (with an applicable CSB+-Tree index) is nearly equal.*



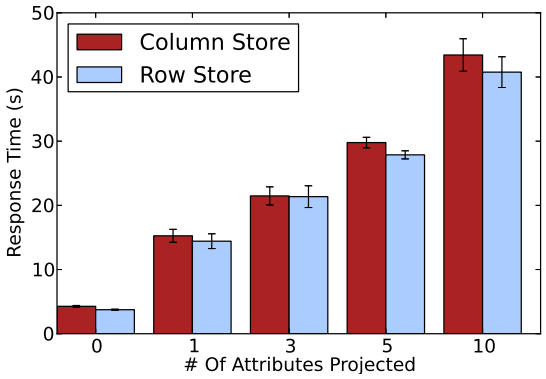
Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	CSB+-Tree
Predicate	1% Selectivity on Indexed (Non-Sorted) Column
Projection Width	Varies

Figure 3.6: Column-Store vs. Row-Store – Indices

When testing queries that include a predicate on a column other than the column-store's sorted column, it was found that, when using blocks, row-stores outperform column-stores for predicates that select a small number of tuples and benefit from using an index, as seen in Figure 3.6. When accessing column values to perform the projection, all values lie on one or two adjacent cache lines in the row store, whereas the each column value in a tuple is on a different (non-contiguous) cache line in the column store. The performance advantage for row-stores is larger for wider projections because column stores incur more cache misses for each additional column in the projection, while row stores do not. In the example shown, when projecting all 10 columns, 97.6 million L3 cache misses are incurred when using a column store, but only 22.0 million when using a row store.

For predicates that select a large number of tuples and are better evaluated with a scan, the performance of row stores and column stores is equal, as seen in Figure 3.7. When scanning, the access pattern is linear and prefetching is effective at avoiding cache misses (both for the single tuple-storage region in the row-store and the densely-packed values in column stripes in the column store).

Observation 7. *For queries with a predicate on a non-sorted column, with a* ***block-based*** *organization, row-stores outperform column-stores when using indexing. This result holds for various*



Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None
Predicate	50% Selectivity on Non-Sorted Column
Projection Width	Varies

Figure 3.7: Column-Store vs. Row-Store – Scanning

selectivity factors and is more pronounced for wider projections. For scan-based queries, row stores and column stores have similar performance.[4]

When these same queries (with predicates on a non-sorted column) were tested using large-file organization, it was found that for queries with 0.1% and 1% selectivity, a row-store with a CSB+-Tree index outperforms a column-store with a CSB+-Tree index for projections of 3 or more attributes. For narrower projections, and for all queries at 10% selectivity, the column store and the row store (both with CSB+-Tree index) have the same performance. Just as in the block-based organization, predicate evaluation using an index takes the same amount of time in either case, but row-stores benefit from the fact that all column values for a projection lie on one or two adjacent cache lines when projecting more than one attribute. For queries with 50% and 100% selectivity (where indices were no longer useful) it was found that the column-store slightly outperforms the row-store for projections of 1, 3, or 5 attributes, and that the row-store outperforms the column store for projections of 10 attributes. Column stores perform better at high selectivity factors because, when scanning, column stripes are accessed in a prefetching-friendly linear pattern, and when selecting 50% or 100% of tuples, there tend to be several matching attribute values on every single cache-line accessed (8 values/line on average at 50% selectivity). Row stores do better when projecting all 10 attributes because a single contiguous row can be directly copied rather than being reassembled from several disjoint column stripes.

Observation 8. *For queries with a predicate on a non-sorted column, with a **file-based** organization, whether row-stores or column-stores perform best is situational and dependent on selectivity,*

[4]Note that, at 50% selectivity (where it is more efficient to evaluate predicates with a scan than an index), column-stores outperform row-stores for wide columns and for narrow projections of wide rows.

projection width, and indexing.

As a point of comparison, scans of column-stores vs. row-stores were also tested in a leading commercial main-memory analytical database, which is referred to as DB-X. DB-X supports both types of storage. The relative performance of column-stores and row-stores in DB-X is similar to what was observed in files in Quickstep (i.e. column-stores slightly outperform row-stores when scanning).

Queries comparing row-store and column-store performance at larger block sizes (64 MB and 256 MB) were also tested to see if, at large block sizes, blocks start to behave more like files. 64 MB blocks have results similar to Observation 7. For 256 MB blocks, when using an index at lower selectivities, column-stores perform similar to row-stores for narrow projections, while row-stores perform best with wider projections. When scanning at large selectivity factors, column-stores perform best when projecting 1 attribute, while row-stores perform best when projecting all 10 attributes (performance is nearly equal when projecting 3 or 5 attributes). The relative performance of row-stores and column-stores in large blocks (256 MB) is similar to that in files.

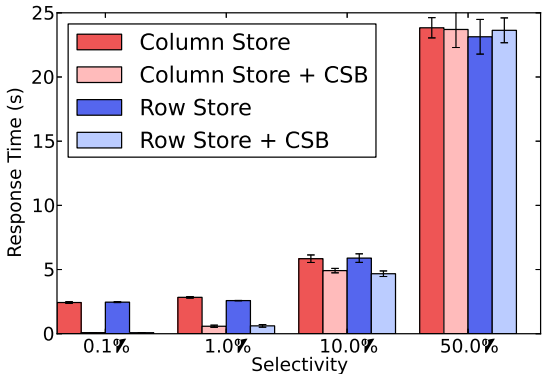
Row-stores and column-stores have essentially identical memory footprints (there is exactly the same amount of data, the only difference being whether it is organized in row-major or column-major order). The Narrow-U table (without indices) takes up 28610 MB in files and 28624 MB in blocks for both column-store and row-store layouts.

Effect Of Indices

As noted above and illustrated in Figures 3.4 and 3.5, when evaluating a predicate that selects based on the value of the sort column in a column-store, a column-store tends to slightly outperform indexed access. This section explores the effect of indexing on row-stores, and when evaluating a predicate on a non-sorted column of a column-store.

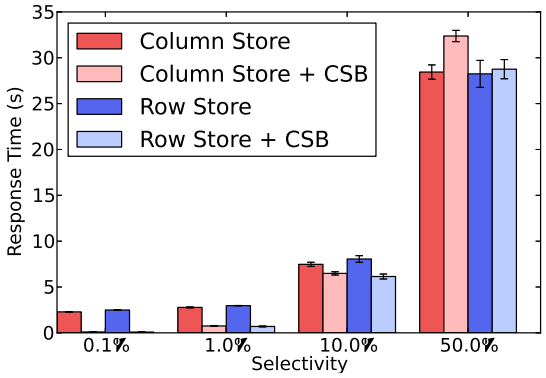
Figure 3.8 shows the effect of using a CSB+-Tree index sub-block to evaluate a predicate where the underlying tuple-storage sub-block is a row-store or column-store. At 0.1%, 1%, and 10% selectivity, using the index is faster than scanning the tuple-storage sub-block. At 50% selectivity, using the index is roughly equal in performance to a scan. Results for 100% selectivity are not shown, as queries that match all the tuples automatically skip any indices. Similar results were observed for other projection widths.

Figure 3.9 similarly shows the effect of using a CSB+-Tree index in the large-file layout. Using an index again reduces query run time at 0.1%, 1%, and 10% selectivity, whether



Organization	Blocks
Tuple Storage Layout	Column-Store & Row-Store
Indexing	**None vs. CSB+-Tree**
Predicate	Varying Selectivity on Indexed (Non-Sorted) Column
Projection Width	3 Columns

Figure 3.8: Effect Of Indices (Blocks)



Organization	File
Tuple Storage Layout	Column-Store & Row-Store
Indexing	**None vs. CSB+-Tree**
Predicate	Varying Selectivity on Indexed (Non-Sorted) Column
Projection Width	3 Columns

Figure 3.9: Effect Of Indices (Files)

the base table file is a row-store or column-store. At 50% selectivity, an index is no longer useful, and for column stores it is faster to simply scan the base table.

Using an index outperforms scanning at lower selectivity factors, because predicate evaluation with an index requires only a logarithmic-time traversal of the CSB+-Tree structure, after which only matching tuples are accessed to perform the projection. Scanning requires accessing every tuple in the table individually to check the predicate (a linear-time procedure). In the example shown in Figure 3.8, at 0.1% selectivity, the total number of L3 cache misses is 1.38 million when using an index with a row store vs. 83.1 million when scanning the base row store. At large selectivity factors, most tuples must be accessed anyway, so the advantage of the index is muted. Additionally, scans access data in a purely linear pattern, which allows prefetching to be effective at avoiding cache misses. In the same example, at 50% selectivity, using an index incurs 682 million L3 cache misses, while scanning incurs only 148 million L3 misses (an index still performs competitively de-

spite this cache-miss disadvantage because the predicate does not need to be individually checked for every tuple).

Observation 9. *For queries with a predicate on a non-sorted column, using a CSB+-Tree index improves query performance for selectivity factors below 50%. This result holds for blocks and files, for both column-store and row-store tuple-storage layouts, and across all projection widths.*

These results demonstrate that secondary indices can play a major role in accelerating query performance in a main memory analytic database. The scan-only approach of systems like Blink [16] greatly simplifies query optimization, but excluding indices from the system can unnecessarily penalize the performance of queries with lower selectivity factors. Also note that Quickstep's block-oriented storage allows the global query optimizer to remain index-oblivious, with the decision of whether to use an index being made independently on a per-block basis within the storage system.

Adding a secondary index necessitates using some additional storage. The additional memory footprint of a CSB+-Tree index on a single column of the Narrow-U table is 7152 MB for blocks and 7152.56 MB for files (25% of the storage used for the base table in either case).

Ordering and Cache Behavior

As noted above, at large selectivity factors, using an index can actually increase the number of cache misses compared to simply scanning the base table. The increased cache misses arise because the order of matches in the index is, in general, different from the order of tuples in the base row-store or column-store, resulting in a random access pattern when accessing tuples to perform the projection. It is natural to ask whether it is possible to improve index performance by first using an index generate a list of tuple-IDs matching a predicate, then sorting this list into ascending order before accessing values in the base table to perform the projection. Tuples in the base table are then accessed in their physical order, effectively changing a random access pattern into a linear one, helping to avoid cache misses and take advantage of prefetching.

Experiments were conducted to determine the effectiveness of sorting the matches (i.e. whether improvements in cache behavior are worth the additional time required to sort the matches). The sorting optimization performs best in combination with a column-store in a file-based organization. For example, when projecting 10 columns at a selectivity factor of 10%, sorting matches reduces the total number of L3 cache misses from 824 million

to 430 million. Despite this improvement, the overall query response time remains nearly the same (nearly all of the benefit of ordered access is negated by the cost of sorting). When the table is in a row-store format, or when using block-based organization instead of files, sorting was not as effective at avoiding cache misses, and as a result the overall response time actually increased due to the additional cost of sorting.

Index Build Cost

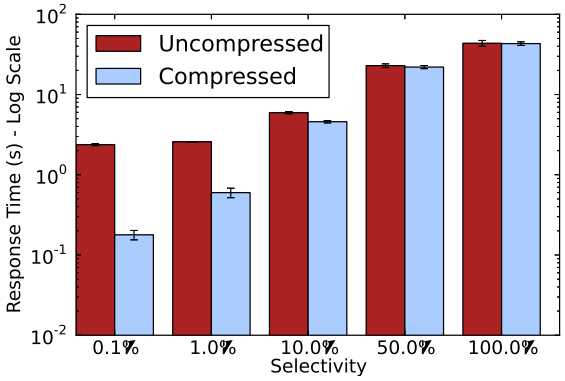
The time to bulk-load CSB+-Tree indices was measured for both the file and the block organizations, with both row-store and column-store tuple-storage layouts. 20 worker threads were used in all cases, with each thread working on one of the static partitions in the file organization, and working on individual blocks in the block organization. For files, the index build time is 111.6 seconds with a row-store and 87.3 seconds with a column-store. For blocks, the index build time is 61.6 seconds with row-stores and 33.6 seconds with column-stores. Index build time is only 38-55% as long for blocks as for files, and 55-78% as long when the base table is a column-store as a row-store. Note that, when using indices on a non-sorted column, performance is better with blocks than with files (see Figure 3.3). Build times are smaller with blocks because the tuple values in relatively small blocks constitute much smaller "runs" to sort into order and build a shallower tree from. Build-times are smaller for column-stores because all of the values that are accessed to build the index are in contiguous regions of memory, efficiently packed with 16 values per cache line.

Observation 10. *The build time for CSB+-Tree indices is much smaller for blocks than for files, even though block-based indices give better read query performance.*

Effect of Compression

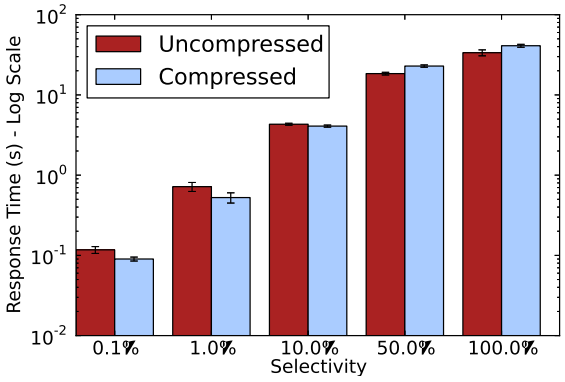
Bit-packing and dictionary-coding compression techniques described in Section 3.3 were tested in combination with row-stores, column-stores, and CSB+-Tree indices on the Narrow-E table. In general, columns 1 and 2 can be represented by single-byte codes, and columns 3, 4, and 5 can be represented by 2-byte codes. Compression reduces the size of the Narrow-E table in blocks from 28624 MB to 20032 MB (compression ratio is sensitive to data distribution, and is likely to be quite different for different tables). Predicates on a compressed column (column 5) were tested, and the projected attributes were randomly varied as in other experiments.

Generally, compression improves performance at selectivity 10% and below, and worsens performance at selectivity 50% and above. The performance effects of compression are consistent for column-stores (with predicates on sorted or unsorted columns) and row-stores, with and without indexing, for all projection widths. Compression speeds up predicate evaluation, as predicates can be evaluated directly on compressed codes and all storage formats are more densely packed in memory, leading to more efficient usage of caches and memory bandwidth. However, compression also increases the cost of performing projections, as codes must be decompressed before they are written to (uncompressed) output blocks. When selectivity factors are low, the cost of predicate evaluation is dominant and compression improves performance. When selectivity factors are high, the cost of decompression begins to overwhelm the advantage from faster predicate evaluation.



Organization	Blocks
Tuple Storage Layout	Column-Store
Indexing	None
Predicate	Varying Selectivity on Non-Sorted Column
Projection Width	3 Columns

Figure 3.10: Effect Of Compression (Column Store Scan)



Organization	Blocks
Tuple Storage Layout	Row-Store
Indexing	CSB+-Tree
Predicate	Varying Selectivity on Indexed Column
Projection Width	3 Columns

Figure 3.11: Effect Of Compression (Row Store With Index)

The effect of compression when scanning a non-sorted compressed column of a column-store is illustrated in Figure 3.10. Here, the performance improvement for compression is

most dramatic. Scans can go through small, dense compressed column stripes, avoiding most cache misses thanks to prefetching. In the example shown, for a predicate with 0.1% selectivity, enabling compression on the column store reduced the total number of L3 cache misses from 5.98 million to 3.32 million. The effect of compression when using a CSB+-Tree index with a row-store is also illustrated in Figure 3.11. These results are more typical of the benefit seen from compression (similar patterns for scans of row-stores, and when evaluating predicates on a column-store's sort column, or on a column-store with an index were observed), which modestly improves performance at 10% selectivity and below, and slightly worsens performance at 50% and above.

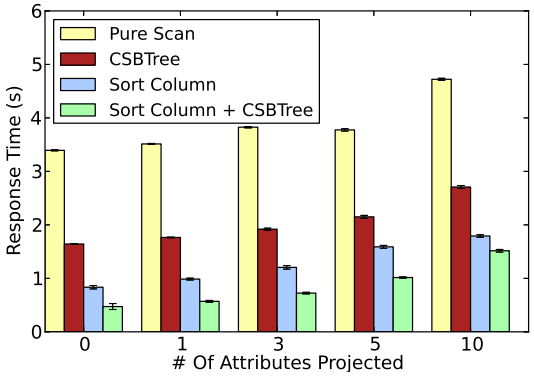
Observation 11. *Dictionary-coding and bit-packing compression improve performance when selecting via a predicate on a compressed column at selectivity 10% or below. Compression makes performance slightly worse at selectivity 50% and above. These results hold for column-stores and row-stores, with and without indexing. Performance improvements are most pronounced when scanning column stores.*

Complex Predicates

This section examines how conjunctive predicates affect the performance of different storage organizations. The predicates in these queries are conjunctions of three single-column predicates, and the projected columns are randomly chosen for each run as in other experiments.

There are a number of strategies possible for evaluating complex predicates, depending on the storage organization. Four simple predicate evaluation strategies are evaluated here across the space of storage organizations studied (note that not all of these strategies are possible in every organization). A pure scan, where values for each predicate column are explicitly read and checked, remains the simplest strategy for both column-stores and row-stores. For a column-store where one of the predicates is on the sort column, a binary search can quickly evaluate that predicate, and the matching tuples can then be scanned and filtered by the remaining predicates on other columns (i.e. a scan, but with a simpler predicate over a smaller range of tuples). When an index is present on one of the columns in the conjunction, the index can be used to evaluate that predicate, and values from matching tuples can be fetched to explicitly check the remaining predicates (again, evaluating a simpler predicate over a smaller number of tuples, but in this case in random order). Synergies between these specialized methods are possible. As a fourth strategy,

the case where one predicate is on the sort column of a column-store and another is on an indexed column is evaluated. The range of tuples that match the first predicate is determined first (using a binary search), then the index is used to evaluate the second predicate and automatically skip over any tuples which aren't in the range for the first. Finally, column values for any tuples which are known to match the first two predicates are fetched to evaluate the third predicate.



Organization	Blocks
Tuple Storage Layout	Column-Store
Indexing	**None vs. CSB+-Tree**
Predicate	1% Selectivity (3 Columns)
Projection Width	Varies

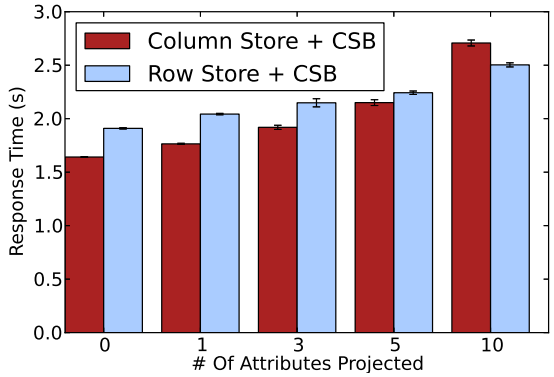
Figure 3.12: Conjunction – Evaluation With Column-Store & Index

The performance of these strategies is illustrated in Figure 3.12. Here it can be seen that a pure scan performs worst, and using either a CSB+-Tree index or a search on the column-store's sort column is effective at reducing query response time. When projecting three columns, the pure scan incurs 38.7 million L3 cache misses. Using a CSB+-Tree index actually increases the number of L3 cache misses to 163 million (the random access pattern is less amenable to prefetching), but it makes up for this by reducing the number of tuples that predicates must be explicitly checked for by 78% (at 1% selectivity), and also reducing the number of predicates that must be explicitly checked from 3 to 2. Doing a binary search on the sort column has all the advantages of using an index, but also has the additional advantage of reducing the number of L3 cache misses to 16.3 million.

The virtuous effects of the index and the sorted-column search compound each other when they are used in combination to evaluate different parts of a conjunctive predicate, significantly outperforming either used alone. The combination of both techniques reduces the number of tuples that must have predicates explicitly checked to just 4.6% of the tuples in the table (at 1% selectivity), and only one predicate needs to be explicitly checked. The number of L3 cache misses incurred when using the combined technique is 39.8 million (about the same as a scan, and worse than just the binary search, but overall

performance is better because there is only one remaining predicate, and it needs to be checked for far fewer tuples). Similar results for conjunctions were observed at other high selectivity factors.

Observation 12. *For conjunctive predicates with overall high (<10%) selectivity, using a sorted-column search and a CSB+-Tree index in combination outperforms either technique used on its own.*



Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	CSB+-Tree
Predicate	1% Selectivity (3 Columns)
Projection Width	Varies

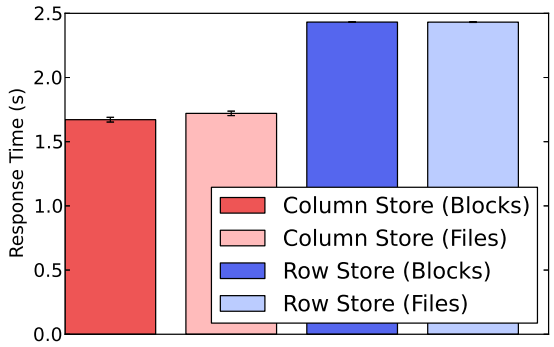
Figure 3.13: Conjunction – Column Store vs. Row Store – Indices

For most conjunctive queries, the previously stated observations regarding the relative performance of different storage organizations hold. One exception to this is Observation 7. When evaluating a conjunctive predicate with a low selectivity factor (0.1% or 1%) using an index on a single column in the predicate, column stores slightly outperform row-stores for narrow projections as illustrated in Figure 3.13. After matches are obtained from the index for part of the conjunction, the rest of the predicate must be checked by fetching values from two other columns in the base table, and these additional values are more densely packed in cache lines in a column-store (in the example shown, when projecting 1 column, the number of L3 cache misses is 202 million for the row-store, but only 148 million for the column-store).

Aggregation

As a final experiment, an aggregate query with a `GROUP BY` clause (with 100 partitions) was run against the different storage organizations studied. A multithreaded hash-based implementation of `GROUP BY` is used, where each worker thread maintains its own hash-table keyed on the value of the grouping attribute, with a payload which is the aggregate's

"handle" (in this case the running value for `MIN()`). The final step in query execution is to merge the per-thread hash tables together into a single global result and apply the `HAVING` condition as a filter. Response time for aggregate queries is shown in Figure 3.14.



Organization	**Blocks vs. Files**
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None
Predicate	None
Projection Width	N/A

Figure 3.14: Aggregate – `MIN` with 100 partitions

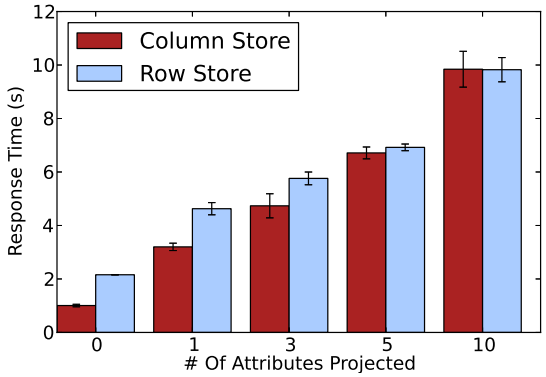
In general, the performance of aggregation closely tracks the performance of a scan with a two-column predicate and no projected output. For every tuple, two columns values are read: the group-by column, which is used to probe the hash table, and the aggregated column, which is compared with the running minimum for a given partition and possibly replaces it. The hash tables for this 100-partition query are small and easily fit in the L1 data cache, so the dominant cost is the cost of reading all the values from two columns of the base table. Prefetching is effective for this linear-scan access pattern, and the dense packing of values in the column store causes it to incur fewer cache-misses and outperform the row store in this case (the block-based row-store incurred 172 million L3 cache misses, but the block-based column-store incurred only 10.8 million).

Efficiently computing aggregates in-memory is an active area of research, particularly when data cubes and advanced holistic aggregates are involved [77, 90]. A full study of main-memory aggregation is beyond the storage-engine focus of this chapter, but the results do indicate that for simple aggregate operations, aggregate performance is closely related to scan performance.

Effect of Wide Columns

In order to study the effect of wide columns, the experiments from previous sections were repeated using the Strings table, which has ten 20-byte wide columns and 1/5 the number of tuples as the Narrow-U table. Most of the queries tested experience a reduction in

runtime on the order of 3X to 5X compared with the equivalent queries on the Narrow-U table, which shows that table cardinality is a dominant linear factor in response time.



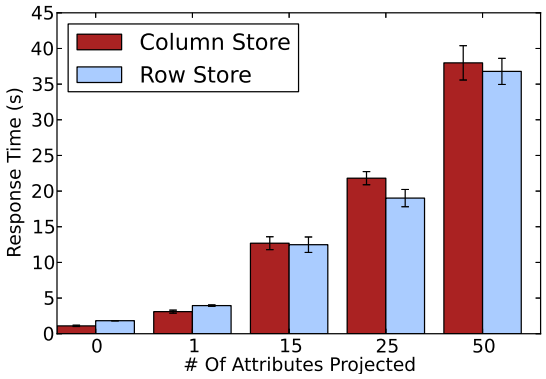
Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None
Predicate	50% Selectivity
Projection Width	Varies

Figure 3.15: Wide Columns (Strings Table) – Scanning

With the overall reduction in query response time, results for the Strings table are remarkably consistent with those for Narrow-U. The previously identified observations hold, with the following caveat: in the block-based organization, column-stores are more competitive with row-stores for queries with a predicate on a non-sorted column. At 0.1%, 1%, and 10% selectivities, when using an index, row-stores still outperform column-stores when projecting more than one attribute, but the difference is less pronounced. At 50% selectivity, when using a scan, column-stores outperform row-stores for narrow projections. This result is illustrated in Figure 3.15. This behavior is because tuples in the strings table are 200 bytes wide and span over 4 or 5 cache lines in a row store, which mutes the advantage row stores have when projecting small values for several columns which were often on the same cache line in Narrow-U.

Effect of Wide Rows

In order to study the effect of wide rows, the experiments from previous sections were repeated using the Wide-E table, which has 50 integer columns and 1/5 the number of tuples as the Narrow-U table. Most of the queries tested have a runtime similar to the equivalent queries on Narrow-U when producing the same volume of output (i.e. 1/5 as many tuples but 5 times as many columns). Results for the Wide-E table were also consistent with those for Narrow-U. All previously identified observations hold for wide rows, except that in the block-based organization, at selectivity factor 50% (at which point indices are not useful, and predicates are evaluated via a scan), a column-store is faster



Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None
Predicate	50% Selectivity
Projection Width	Varies

Figure 3.16: Wide Rows (Wide-E Table) – Scanning

for narrow projections, while a row-store is faster for wide projections, as illustrated in Figure 3.16. Again, this can be partially attributed to the fact that wide 200-byte rows are spread across 4 or 5 cache lines in a row store, and it is necessary to project a substantial portion of the columns to see a benefit from locality of access due to several projected attributes from a matching row lying on the same cache line. Additionally, a column store will typically store values from several matching tuples together on the same cache line for a query with high (~50%) selectivity, so that a cache line which is fetched to perform a projection for one tuple will typically remain resident and also be used when projecting the same attribute for the next few tuples. For example, in the figure shown, when projecting one column, the row-store incurs 241 million L3 cache misses, but the column-store incurs only 3.72 million. Note that for smaller selectivity factors, when using an index, a row-store still outperforms a column-store across the board in the block organization.

3.5 Summary of Experimental Findings

The experimental observations in this chapter can be distilled to the following insights for physical organization of data in a read-optimized main memory database:

1. **Block-based organization should always be used over file-based organization**. Performance for blocks is always at least as good as files, and often better. Additionally, blocks have cheaper load costs, both for sorting column-stores and building CSB+-Tree indices.

2. **Both column-store and row-store organization should be available as options for**

tuple-storage layout. Column-stores perform best when there is a single dominant (i.e. sorted) column which predicates select on, or when the selectivity factor is large (~50%) and the columns are wide, or rows are wide and projections are narrow. Row-stores perform best when selecting via predicates on various different columns, with or without an index, except for edge cases at large (~50%) selectivity factors noted previously. These results show that despite the fact that most leading analytic DBMSes are purely or primarily column stores, there is a significant class of queries where row-stores have the performance edge.

3. **CSB+-Tree indices, co-located with data inside blocks, are useful in speeding query evaluation with predicates on a non-sorted column**. The advantages of CSB+-Tree indices are applicable whether the tuple-storage format is a row-store or column-store, for selectivity factors smaller than ~50%. At larger selectivity factors, it is better to simply scan the base table. This shows that indexing can still play a role in improving query performance in the in-memory setting, and in-memory database designs that are purely scan based (e.g. BLINK [85]) miss an opportunity to apply a technique that can significantly reduce query response time, particularly for highly selective queries.

4. **Compression can improve query performance across the other dimensions of storage organization studied.** When a column's data is amenable to compression, compression consistently improves performance for queries at 10% selectivity or below, especially when scanning an unsorted column of a column store. On the other hand, compression can slightly diminish performance at higher selectivity factors. Previous studies [51] have emphasized the benefits of compression in speeding up predicate evaluation, and compression is widely deployed in commercial analytical DBMSes [35, 65, 84, 85, 92]. The results in this chapter demonstrate that some care must be taken when applying compression, however, as the cost of decompressing attribute values when performing compression can outweigh the advantages from faster predicate evaluation when selectivity factors are high.

3.6 Related Work

The design of high-performance main memory databases to support analytical workloads has been a vibrant area of research for over a decade. An early pioneer in this area (and

also in the field of column-stores) was MonetDB [25], and subsequently Vectorwise [113]. Quickstep applies lessons from MonetDB and Vectorwise, particularly in how it materializes intermediate results using tight execution loops.

There are a number of commercial products that target the main memory data analytics market, including SAP HANA [35], IBM Blink [16, 85], Oracle Exalytics [75], and Oracle TimesTen [64], as well as academic projects like HyPer [60] and HYRISE [46]. These systems vary widely in terms of storage organization, with some employing row-stores, some column-stores, and some both, and each system offering different options for compression and indexing.

The space of possible data organizations is not limited to row-stores and column-stores, and includes other alternatives such as PAX [3] and data morphing [47]. The HYRISE project [46] has adapted data morphing techniques to the main memory environment. To control the scope of the experiments presented here, only the widely-used row-store and column-store layouts were studied in this chapter. It should be noted, however, that the modular and flexible Quickstep storage system (see Section 3.2) makes it easy to integrate new storage formats into Quickstep and conduct similar experiments with them.

3.7 Conclusion

This chapter identified and evaluated key parts of the design space of storage organizations for main memory read-optimized databases that can have a major impact on the performance of analytic queries. The empirical evaluation presented here has found that block-based organization performs better than file-based organization, that column-stores and row-stores each have advantages for certain classes of queries (and both options should be considered for high-performance), that CSB+-Tree indices, co-located with data inside blocks, can play a major role in accelerating query performance, and that where data is amenable to compression and selectivity factors are sufficiently small, compression can also improve query performance.

The design space for storage organizations is large, and while the key dimensions of the design space are explored in this chapter, there are also other open areas that are worth exploring. The Quickstep storage manager produced as part of this work could be used to explore portions of the design space that are not covered in this study.

Perhaps the most important overriding result in this chapter is that in most dimensions of in-memory storage organization, there is no "one size fits all" design choice that

consistently achieves the best performance across a range of queries. For instance, despite the fact that a column-store is the default or only tuple storage layout available in many leading analytical databases [65, 84, 92, 113], row-stores still manage to outperform column stores in many queries where several attributes are projected due to superior cache locality properties. While some systems like BLINK [85] eliminate indices entirely and evaluate all selections with scans, the experiments in this chapter showed that CSB+-Tree indices can significantly reduce query evaluation time, especially for highly-selective queries. Which storage organization choices are most efficient depend on the data and the query load, and as such a *flexible* storage engine that allows different physical data organizations to coexist within the same system is useful. Physical database design in such an open environment is a challenge, but the specific experimental results in this chapter are a useful basis for both manual physical schema tuning and the development of automated tools to choose physical organization parameters.

4 TRANSACTIONAL MESSAGE BUS: A RELIABLE AND SCALABLE COMMUNICATION PARADIGM

4.1 Introduction

A crucial component of any scalable distributed system is a communication fabric that allows different actors in the system to communicate with each other. For example, a distributed search engine needs to send queries to different workers (each in charge of a distinct partition of the data) in the system, and then collect these results. A distributed data processing system needs a query coordinator to send sub-queries to different workers in the system and aggregate results.

Even in a single-process version of Quickstep, communication between the scheduler and many worker threads is an important concern. Simple mechanisms like a global FIFO work queue do not scale well with many cores, especially in a NUMA setting where access to shared memory often requires reading from non-local NUMA memory (which is slower than local memory both in terms of latency and bandwidth) and generates a large amount of cache-coherency traffic (and cache evictions) between sockets. The Transactional Message Bus described in this chapter serves an important role for in-process communication, and can also be used transparently over the network in a future distributed version of

Feature	TMB	ActiveMQ	Spread Toolkit	Kafka	Akka	Amazon SQS	ZeroMQ
Point-to-point Messaging	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Group Messaging	Yes	Yes	Yes	Limited▲	Limited*	No	Limited*
Guaranteed Delivery	Yes	Optional	Yes	Yes	No	Yes	No
Persistent Queues	Yes	Optional	No	Yes	Optional	Yes	No
Queryable Queues	Yes	Yes	No	Yes	No	No	No
Virtual Synchrony	Yes	Optional✢	Yes	No	No	No	No
Robust to Client Failures	Yes	Yes	Partial♦	Yes	Yes	Yes	No
Robust to Service Failures (highly available)	Yes	Optional❍	Yes	Yes	N/A	Yes	N/A
Tied Messages	Yes	No	No	No	No	No	No
Priority & Deadlines	Yes	Yes	No	No	Deadlines Only	Deadlines Only	No
Message Size Limit	None	None	100 KB	Varies	Varies	256 KB	None

▲ Publish subscribe or "consumer group" that routes each message to single recipient only.
* Publish-Subscribe Only
✢ Virtual synchrony only available in single-broker or master-slave configuration.
♦ Messages can be lost if a client crashes during receive and later recovers.
❍ Highly available only in multi-broker or failover configuration.

Table 4.1: Messaging Framework Feature Comparison

Quickstep.

In early distributed systems, communication was often ad-hoc and not cleanly abstracted into a framework, which made programming and reasoning about concurrent behavior difficult and error prone, especially in the presence of unreliable components and networks. These challenges motivated the development of reusable general-purpose communications systems, and the history and development of several notable examples are covered in Section 4.2. Today, "Message-Oriented Middleware" (or MOM) is a major industry, with many different products available that serve as key building blocks for diverse distributed applications. MOM systems have also been of particular interest to the database community, since message queues can be viewed as a type of database and the operation of a MOM system as a special case of transaction processing.

Surprisingly, there has not been a systematic evaluation of such communication frameworks. This limitation is addressed in this chapter. There are many good ideas and features in existing communication frameworks, but there isn't a single system that provides a comprehensive set of desirable features. Thus, this chapter introduces a new communication framework called a Transactional Message Bus, or TMB. Table 4.1 compares the TMB's feature set with that of popular communication frameworks, including Apache ActiveMQ [7] (a popular message broker implementing the Java Message Service [48]), the Spread Toolkit [94] (a virtually synchronous messaging framework), Apache Kafka [8] (a distributed, partitioned message logging service), Akka [100] (a Java-based toolkit for building message-driven distributed applications), Amazon Simple Queue Service [6] (a cloud based persistent queue service), and ZeroMQ [54] (a lightweight embedded broker-free messaging library). None of these existing systems have the *entire* ensemble of features present in a TMB, which this chapter will argue is needed to build distributed systems that are scalable, performant, and reliable.

The second contribution of this chapter is to cleanly define the semantics of TMBs (see Section 4.3). TMBs are *transactional* in that the sending and receiving of asynchronous messages are ACID transactions with guaranteed delivery, data persistence and recovery support, and a consistent, deterministic set of semantics for addressing and ordering messages based on the well-known model of virtual synchrony [19] with some extensions, such as application-defined message priority and cancellable messages. Although the message-level transactions provided by the TMB do not automatically translate to application-level ACID transactions in a distributed system, consistent and reliable messaging semantics can make it easier to implement application-level transactions.

The last contribution of this chapter is the design and evaluation of a "pluggable" modular software architecture for TMBs that allows components providing transaction management, durable persistent storage for messages, and network transparency to be freely combined into a complete TMB stack. Building on, and inspired by, the observation that communication between workflows via persistent queues is a database problem [44], these TMB components have been implemented using a variety of both relational and NoSQL database systems (SQLite, VoltDB, Zookeeper, and LevelDB). Additionally, lightweight "native" components have been implemented at each level of the stack that are written from scratch and do not depend on any third-party database. A key contribution of this chapter is considering how a full-featured messaging framework can be built from various transaction processing systems, and what implications different approaches to transaction processing have for the reliability, performance, and scalability of messaging. Section 4.4 describes the design and tuning of these different TMB implementations in detail. Section 4.5 contains a comprehensive experimental evaluation of the performance of these implementations. Section 4.5 also presents an experimental comparison of the performance of TMBs with Apache ActiveMQ [7] and the Spread Toolkit [94], two popular purpose-built MOM systems.

4.2 Related Work

Communication is crucial to any distributed or parallel application, but it is impractical and error-prone to design a custom communication subsystem from scratch for every such application. As such, many distributed systems rely on a common reusable communication service that provides a useful abstraction for communication among many actors while masking the complexity of the underlying implementation. There have been many efforts to integrate rich feature-sets and strong guarantees about consistency and reliability into such communication frameworks in order to make programming applications easier and reduce the need to develop ad-hoc solutions to problems that can arise in an unpredictable and sometimes unreliable network environment. This section describes some of this related work.

Classical Communication

One of the earliest abstractions for communication is the remote procedure call [22] (RPC), which allows invoking code on remote machines. More recent methods include Message

Passing Interface (MPI) [45] to program distributed-memory systems, and CORBA [50, 102] to access "distributed objects." These methods cover some, but not all, features in TMBs. Notably, the notion of reliability of RPC has been addressed by systems like ISIS [20], which introduces the "virtually synchronous" execution model that is the basis for the TMB's semantics. There is also work on pushing reliability and consistency features to the communication layer in CORBA [86] to ease application programming (the TMB is based on a similar design philosophy).

Virtual Synchrony

Virtual synchrony provides distributed processes the illusion of a serial, synchronous sequence of events that occur in the same order for every process. The key aspects of virtual synchrony are address expansion, and delivery atomicity and order [19]. Address expansion means that sending a multicast message to a named group of processes requires all participants to have the same consistent view of group membership when the message is sent and when it is delivered. Delivery atomicity requires that all of the recipients of a multicast message will eventually receive or (only if the sender fails) none do. Finally, delivery ordering may be either causal or absolute. Causal ordering merely requires that for any two messages sent by the same sender to the same (or overlapping) receiver(s), the messages are received in the same order that they are sent (this corresponds to Lamport's definition of the causal happens-before relation [67]). Absolute ordering extends this requirement by imposing a total order on the receipt of messages for all processes in a group. By default, TMBs provide virtual synchrony with at least causal message ordering. TMB implementations that are based on relational database management systems (DBMSs) also provide absolute ordering. TMBs also provide mechanisms whereby applications can intentionally violate causal ordering so that, for instance, higher-priority messages arrive before lower-priority ones.

Message-Oriented Middleware

MOM is a major industry [30], with products such as [6, 7, 52, 81, 82], and standards like the Java Message Service [48] which provides a common API for many products.

MOM systems provide an asynchronous messaging service for cooperating applications. The MOM manages a queue of incoming messages on behalf of each client, and the

acts of sending (enqueueing) and receiving (dequeueing) a message are decoupled and asynchronous.

The basic MOM model has some limitations that have inspired extensions to integrate messaging with distributed ACID transactions [74, 98] and to integrate enforcement of logical conditions on message delivery into the MOM itself [97]. In this body of work, a key focus is to "push down" features like at-least-once message delivery, deterministic ordering, and failure recovery to the communications layer to make it easier to engineer distributed applications. Systems like Horus [101] have even made such additional functionality "pluggable" so that applications can select for themselves which features they need from the messaging layer. This body of work has provided many cues in developing the TMB, which also provides point-to-point and group-oriented messaging that is integrated with strong reliability, consistency, and ordering guarantees as detailed in Section 4.3.

Persistent Queues

A major inspiration for the TMB was work on persistent queues. As the MOM industry developed and began to integrate with DBMSs, it was realized that message queues are themselves transaction-processing DBMSs, and can be implemented using existing relational DBMS features [44]. Some MOM advocates countered that existing DBMS products were too "heavyweight" and slow for messaging applications [15], but DBMSs have continued to make performance improvements, especially in main-memory systems [53, 59], and experience in developing the TMB has shown that modern transaction-processing databases are more than up to the challenge.

The integration of transaction processing (TP) monitors and DBMSs with message-driven distributed systems also lead researchers to realize that to build a truly reliable TP system requires a persistent, recoverable system for request queues [18]. Reliable queueing of requests has been integrated into leading commercial DBMS products as part of a service-oriented database architecture [28]. TMBs owe a great deal to lessons learned from persistent queues and generalize the concept to many different styles of asynchronous communication suitable for diverse application domains. TMBs also include built-in features such as tied messages, group messaging, and virtual synchrony which are crucial for modern distributed applications; thus, making writing distributed applications with TMB much easier.

4.3 TMB Semantics

This section develops the TMB semantics, including reliable and consistent message delivery even in the face of failures.

Example

To illustrate the TMB API, consider a simple distributed search application as an example. This application consists of a several servers that each contain a different partition of some text data. Clients can search the data by sending a request to each server and collecting all of their responses, using the following TMB pseudo-code:

```
search(keyword):
  for server in search_servers:
    tmb.Send(server, SearchRequest(keyword))
  responses = tmb.Receive()
  return concatenate(responses)
```

The servers run a message-driven loop to service requests:

```
loop:
  request = tmb.Receive()
  matches = find request.keyword in local_data
  tmb.Send(request.sender, SearchResponse(matches))
```

The server-side loop blocks until a message is available, then proceeds to search its local partition of the data, and send results back to the client that made the request. Sending and receiving a message are decoupled and asynchronous, so clients and servers don't have to wait for each other, but the guaranteed reliable delivery and abstract addressing provided by the TMB means that the application code doesn't need to worry about lost messages or retry loops. Thus, writing the application becomes far simpler given the TMB abstraction.

TMB API

The calls in the TMB API are as follows:

- `Connect()/Disconnect()`: Connects a "client" (i.e. some actor using the TMB) to the bus so it can start sending and receiving messages, and permanently disconnects it, respectively. The `Connect()` call returns a unique ID that the client uses to identify itself to all other API calls.
- `RegisterClientAsSender()/RegisterClientAsReceiver()`: Informs the TMB that a client is capable of sending or receiving a certain type of message. TMBs support sending any number of different application-defined classes of messages, which is discussed in detail below.
- `Send()`: Send a message to one or more other clients of the TMB. The arguments to this call include the message itself (tagged with a type identifier) and an address which specifies recipients. Other optional arguments allow a sender to use additional messaging features that are described below.
- `Receive()`: Receive pending messages. This method is available in both a blocking version that waits until at least 1 message is available and a non-blocking version that returns immediately if no messages are pending for a client. These methods can be used to receive messages one at a time, or to get multiple messages in a batch.
- `DeleteMessages()`: Erase one or more received messages from the TMB. By default, `Receive()` does not erase messages as they are received, so that if a client fails or experiences some error, it can recover and not lose any messages. An explicit call to `DeleteMessages()` can be issued when a client is actually finished processing a message and no longer needs the TMB to retain it.
- `CancelMessages()`: Cancel a previously sent message, preventing any client from receiving it in the future. This call can be made by the client that originally sent a message, or by any of several clients that receive the message. Cancellation is discussed in detail below.

Every TMB API call is implemented as an ACID transaction on the TMB's state (note that this does NOT automatically mean that applications running on the TMB are transactional, but it does help to reason more easily about concurrency and ordering). Below, the semantics of these transactions are discussed in more detail.

Clients

A client is an abstract entity that sends and receives messages using a TMB. Depending on the structure of an application, clients may be independent threads in a parallel pro-

gram, independent processes running on separate machines in a distributed setting, or some other application-specific entity (for example, nodes in a graph-oriented processing model like Bulk-Synchronous Parallel). A client registers itself with the TMB by calling `Connect()`, which returns a globally unique identifier that the client uses to identify itself for any other call to the TMB API.

A TMB "remembers" a client and retains any metadata and pending messages until that client explicitly disconnects by invoking the `Disconnect()` API call. This means that even if a client fails and later recovers (for instance as a result of crash or hardware failure), no messages are lost, and other active clients may still receive messages that the client sent before failing, as well as send messages which a failed client can receive once it recovers. In this way, the TMB provides highly available messaging even though clients may be unreliable.

Messages

Messages are the basic unit of communication between clients. There are no restrictions on the content or format of messages. A message is simply an arbitrary sequence of bytes that are opaque to the TMB itself. From the application's perspective, this abstraction allows virtually any serializable data structure to be a message, including text strings, flat programming language variables and structures, or any of several popular interoperable formats for structured or semi-structured data, such as JSON, XML, Protocol Buffers, and Apache Thrift.

It is natural that applications may use a TMB to send many different "types" of messages, and that different clients may be capable of sending or receiving only certain types of messages. Each message has a "message type" identifier that is specified by the sender. Clients can register as senders or receivers for a particular message type, and the TMB enforces the policy that a client may only send a message of a type for which it is registered as a sender, and that any explicitly-specified recipients of a message must be registered as a receiver of the type. These tests are performed immediately as part of the `Send()` transaction, aborting and returning an error code to the sender if the check fails.

Say that the example distributed search application defines `SearchRequest` as message type 0 and `SearchResponse` as message type 1. A client connecting to the TMB for the first time has the following start-up procedure (the server's procedure is the same, with the message types reversed):

```
my_id = tmb.Connect()
tmb.RegisterClientAsSender(my_id, 0)    // Request
tmb.RegisterClientAsReceiver(my_id, 1)  // Response
```

Sending Messages

The core purpose of the TMB is to deliver messages between clients reliably, but asynchronously. Asynchronous delivery has positive implications for the performance and scalability of TMBs, since clients generally do not need to wait for each other, as well as for availability, since senders can still send messages to clients that have temporarily failed or are "lagging" (processing messages slowly).

To send a message, a client calls `Send()`, supplying the message (including its type identifier) and an address that specifies one or more clients to receive the message (addressing is described below). The TMB checks that the client is connected and registered as a sender of the specified message type, and that each explicitly specified recipient is registered as a receiver of the message type. If the attempted send operation violates any of these constraints, the transaction is aborted and an error code is returned to the client. On the other hand, if the checks are successful, then a copy of the message (plus some additional metadata, including the client ID of the sender and the timestamp at which the message was sent) is pushed on each receiver's *queue* of incoming messages. As with other operations, pushing a message on a client/receiver queue is an ACID transaction. Indeed, each of the four ACID properties is important to the correctness of the TMB implementation: enqueueing a message must be *atomic* so that no partial, garbled, or misordered messages appear, *consistent* so that clients always see a valid queue state and that messages appear in the correct order (see Section 4.3 for details on ordering), *isolated* so that multiple concurrent senders and the receiver itself do not interfere with each other when enqueueing or dequeueing messages, and *durable* so that once a message is sent it is guaranteed to eventually be received by a client so long as that client (or a replacement that is brought up for it after a failure) continues to receive messages. [1]

[1] There are two cases where a message might not be delivered. The first is when client permanently disconnects from the TMB by calling `Disconnect()` without first emptying its queue, in which case "left-over" pending messages are discarded. `Disconnect()` can be made conditional so that it only succeeds if the queue is empty, but applications should still be aware of this edge case. The second case is when a client fails and is never recovered or replaced, in which case messages will stay durably queued indefinitely in anticipation of *eventually* being read or explicitly discarded.

Addressing Modes

A TMB provides two ways of specifying which clients should receive a message. The first is *explicit* addressing, where a sender simply specifies a list of unique client IDs that should receive the message. A client may know these client IDs as a result of out-of-band communication, or as a result of previous communication using the TMB (recall that the ID of the client that sent a message is provided to each client that receives it). To validate an explicit address, the TMB checks that each specified client is connected and, if so, if it is registered as a receiver of the message's type.

The second mode of addressing is *implicit*, where the sender simply requests that a message be delivered to any client that is capable of receiving it (i.e. to any client which is registered as a receiver for the message's type). If no connected clients are capable of receiving the message, then an error code is returned to the sender, otherwise the send transaction proceeds as normal using the set of receivers for the message type as its list of recipients.

Both modes of addressing allow for more than one recipient to be specified for a message. If the sender specifies that a message should be *broadcast*, then a copy of the message will be enqueued for every specified recipient. If the message is non-broadcast, then a single client is chosen from the set of possible recipients to receive the message.

Combining implicit addressing and broadcast allows the clients in our distributed search example to "discover" and send a search request to all servers, as follows:

```
status = tmb.Send(ANY, BROADCAST,
                  SearchRequest(keyword))
if status == NORECEIVERS:
  return {empty}
```

The combination of implicit addressing and broadcast also enables publish-subscribe style messaging in a TMB by simply using a different message type ID for each channel.

Receiving Messages

Each client that is connected to a TMB instance has a persistent, transactionally-consistent queue of incoming messages. The `Receive()` transactions observe a consistent snapshot of the queue and retrieve messages from it. In order to amortize the costs associated with accessing the queue, a client may choose to receive a batch of messages all at once, with

an optional limit on the size of the batch. Both blocking and non-blocking versions of the `Receive()` call are provided. A client that operates a purely message-driven main loop (like the server in the example) would prefer the blocking version, since it has nothing to do without any messages, and simply waiting for the blocking call to return is more efficient than repeatedly polling the non-blocking version. On the other hand, a client may not wish to block waiting for messages while none are available, instead performing some other useful work. In that case, the client can use the non-blocking version that returns immediately if it sees that no incoming messages are currently queued.

Deleting Messages

When and how messages are removed from queues is an important consideration for the reliability of an application that uses a TMB. Messages could be removed from queues when they are received as part of the same transaction. However, this approach can cause problems for some applications when the client fails, which will now be illustrated. Consider receiving and immediately deleting a message from the TMB's durable store as soon as a client receives it. Next, assume that the client fails before it can actually process the message. Later, when the failed client restarts and is ready to receive messages, it will simply start processing new messages. The message(s) that it received, but did not process before the failure event, will seem to have "disappeared". Thus, a client that fails can cause a violation of the durability and guaranteed message delivery properties.

To address this problem, receiving and deleting messages are *separate* operations in a TMB. `Receive()` is a read-only transaction that retrieves a message from the appropriate queue, but leaves the message in-place. Once a client has processed a received message and handled it in the appropriate, application-specific way (including possibly sending out responses or other additional messages using the TMB), it then makes a separate explicit call using the `DeleteMessages()` API. This second call triggers a separate transaction that actually erases the message from the queue. A client that makes multiple `Receive()` calls without deleting messages will see the same messages again, thus ensuring at-least-once delivery of messages even when clients fails. TMBs optionally allow messages to be deleted as they are received (admitting the anomaly discussed above for applications that can tolerate it), but this is not the default behavior.

Say that the example distributed search application requires 100% recall, but servers sometimes suffer temporary outages (the frequency of which naturally increases in a distributed setting). The application can be made robust to such outages by writing the

server's main loop as follows:

```
loop:
  request = tmb.Receive()
  matches = find request.keyword in local_data
  tmb.Send(request.sender, SearchResponse(matches))
  tmb.DeleteMessage(request);
```

Now, if a server fails after calling `Receive()`, but before calling `Send()`, it will reenter the loop when it recovers, receive the request again, and proceed as it normally would. Note that there is a small window where the server could fail after calling `Send()`, but before calling `DeleteMessages()`, in which case the server would do its local search and send a response again. A client-supplied request ID can be added to the request and response messages so that clients can discard such duplicate responses, achieving exactly-once semantics even in the presence of failures.

Additional Messaging Features

Thus far, messages have been treated as opaque, aside from their type. This section develops some additional, but optional, features of messages that can make TMBs more useful in various application settings.

Deadlines & Priority Levels

It is often a case that a sender may only want a message to be received within a certain time frame, especially in interactive applications. In the distributed search example, if some servers are lagging, then it may be preferable to return partial results to the user within a limited time frame rather than to wait for all the servers to respond [31]. After that window has passed, any outstanding requests are obsolete, and processing them is a waste of time on servers that were already lagging (this can be a vicious cycle, as servers that already have long queues have to do *more* work to catch up).

In order to avoid doing redundant work, which compounds the problem of lagging, the TMB allows an optional expiration time to be specified with the `Send()` call. In the corresponding `Receive()` call, each message's expiration time is checked against the timestamp of the transaction, and expired messages are silently discarded. Expiration times also affect the order in which messages are received by clients. Messages that expire sooner are received before those that expire later (messages that have no expiration time are received

last). Thus, the TMB prioritizes messages with an earlier deadline so that, if possible given time constraints, all messages will be received and processed.

The TMB can also give applications explicit control over the relative ordering of messages by allowing senders to specify a priority level for each message. Message queues are ordered in descending order of priority, with ties broken using the earliest-deadline-first policy described above. In the example application, there might be multiple classes of clients sharing the same service. Some are interactive and sensitive to latency, while others are doing offline batch-processing. It makes sense to assign a higher priority to the interactive requests so that they get serviced first, with the long-running batch jobs proceeding when the servers are not otherwise busy. This approach generalizes to any number of priority levels, which can make it easier to achieve service level objectives for different groups of clients.

Ordered Delivery & Streaming

The features described above affect the order in which messages are received, which requires treating the queue of incoming messages for each client as a priority queue. Still, unless a sender explicitly and intentionally changes the order in which messages it sends to a particular client should be received by specifying different priorities or deadlines, it is useful for a sequence of messages from a particular sender to a particular receiver to be received in the same order as they were sent, i.e. to provide virtual synchrony with causal ordering [19]. This semantics facilitates streaming of data via multiple discrete messages, making it easier to reason about messaging between concurrent clients [21]. This functionality is easily achieved by using the send timestamp attached to each message to order messages by their send time when priority and expiration time are the same.

Message Cancellation & Tied Messages

Tied messages are an effective technique for dealing with latency variability in large-scale interactive web services [31]. A major source of variable latency in large-scale distributed services is queueing delay on servers. A tied message is a request which is sent to multiple servers that are able to process it. A tied message includes information about each of the target servers. Clients issue a tied request to two or more servers that are capable of servicing it. Once a copy of the message reaches the head of one of the servers' queue, that server will "cancel" the message for its peers, preventing them from receiving it and

doing unnecessary redundant work, while still allowing the client to benefit from having its request serviced by the lowest-latency server.

Tied or cancellable messages are fully supported by TMBs. When a message is sent, the sender may choose to make it cancellable. For such messages, the TMB creates a cancellation "token" that has information to locate and delete copies of the message in each recipient's queue. The cancellation token is attached to each copy of the message, and a copy is also returned to the sender. The sender may cancel a message at any time using the token. Similarly, receiving clients that receive a cancellable message can cancel it, thus preventing their peers from receiving it in the future.

Note that (as with other tied message implementations [31]) there is a small window of time during which it is possible for a client to receive a message that has just been canceled by another actor in the system. This situation is not an error for the TMB, as cancellation is treated as an idempotent operation. Nevertheless, programmers using a TMB should be aware that tied messages do not guarantee only-once delivery and take steps to ensure that multiple clients receiving the same message do not cause an application-level error (e.g. by doing operations that are idempotent and adding the original message id to the response message, ensuring that operations that modify a shared application state are idempotent or commutative or, failing that, coordinating using a distributed locking or commit protocol).

Tied messages can be used to improve the tail-latency of interactive searches in the running example. Say that each partition of the data set is replicated across multiple servers. The client can then broadcast a cancellable message to multiple servers for each replica set, using the pseudocode:

```
search(keyword):
  for server_set in partitions:
    token = tmb.Send(server_set,
                     BROADCAST,
                     CANCELLABLE,
                     EXPIRES(now + 100 ms),
                     SearchRequest(keyword))
  responses = {}
  loop until all received OR 100 ms:
    responses = concatenate(responses, tmb.Receive())
  tmb.CancelMessage(token)  // prevent redundant work
  return responses
```

While the server code looks like:

```
loop:
  request = tmb.Receive()
  tmb.CancelMessage(request)
  matches = find request.keyword in local_data
  tmb.Send(request.sender, SearchResponse(matches))
```

This way, the client benefits from having each partition's search request serviced by the earliest available server, but the amount of work done on the servers is limited by canceling the tied requests when a server starts processing them, and after the client no longer wants any more responses.

Summary of Logical TMB Structure

This section has developed the features and semantics of the Transactional Message Bus. Any TMB implementation consists of a shared and globally consistent state, as well as per-client priority queues of incoming messages. The global state is the set of connected clients, and the set of message types that each connected client is capable of sending and receiving. All transactions modifying the global state have a serializable order. The per-client priority queues of incoming messages support *push* (i.e. send), *read* (i.e. receive), and *delete* (explicit removal or cancellation of messages) operations. The *push* and *delete* operations are atomic and serializable, and the read-only *read* operation observes a consistent snapshot of the queue.

4.4 TMB Implementations

This section discusses the experience of implementing the TMB as a modular service, leveraging some existing database systems as well as developing "native" implementations from scratch.

Modular TMB Architecture

The software architecture of a TMB implementation is broadly divided into three tiers, as shown in Figure 4.1. At the heart of the TMB is a "Transaction/Bus Management" component that implements the full TMB semantics described in Section 4.3 and enforces the

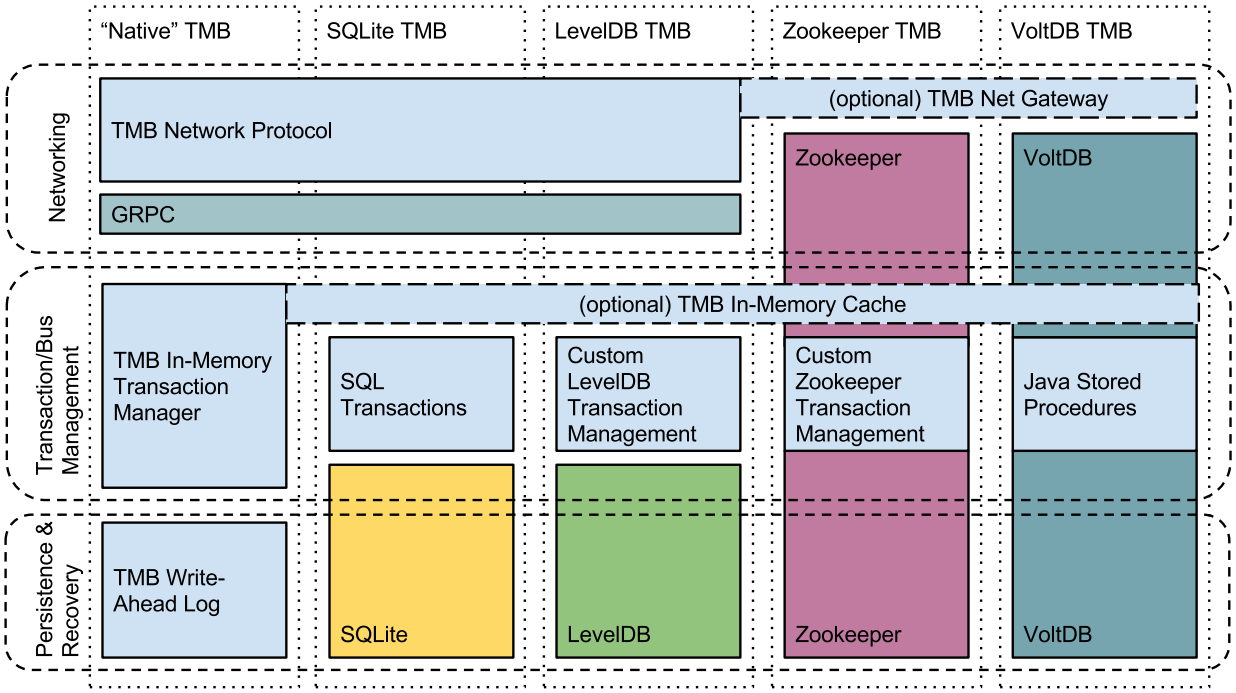


Figure 4.1: TMB Component Architecture

consistency guarantees described there. Below the transaction manager is a durable storage component responsible for storing TMB state persistently and recovering after a crash or other failure. Finally, there is a networking layer that allows TMB clients in different processes running on different machines to share a TMB instance and communicate with each other.

The TMB implementation was designed so that the components in each tier are as "pluggable" as possible. For instance, if a TMB is only being used by clients on a single machine, there is no need for any networking component. Similarly, if there is no requirement for long-term durability of messages, the Native TMB in-memory transaction manager can be used without any persistent storage component as a high-performance in-process "pure memory" message bus.

In the following sections, the different implementations of the three tiers are described in more detail.

Transaction/Bus Management Tier

The bus-management core of the TMB is implemented by a custom in-memory transaction manager, as well as systems of transactions leveraging the underlying features in SQLite,

LevelDB, Zookeeper, and VoltDB. The native in-memory transaction manager can also be used as an in-memory cache in front of any of the four third-party databases that have been experimented with, in effect using the highly-tuned in-memory transaction manager in combination with the durability and recovery afforded by the underlying database's storage system.

Custom In-Memory Transaction Manager

This section describes the custom in-memory transaction manager, which can be combined with any of several options for persistence and recovery, or used alone as a "pure memory" message bus.

Data Structures The global set of connected clients is represented by a hash table that maps unique client IDs to per-client records. The per-client records, in turn, consist of a hash table of sendable message type IDs, a hash table of receivable message type IDs, and a priority queue of incoming messages (both max-heap and balanced binary tree-based priority queue implementations have been evaluated). There is also a secondary index that maps receivable message type IDs to client IDs to speed up the resolution of implicit addresses.

Concurrency Control The initial transaction manager design used well-known concurrency primitives to control access to shared data structures. The global hash-table of clients was protected by a read-write lock. Connect and disconnect transactions acquired this lock in exclusive mode, while all other transactions acquired it in shared mode. Similarly, the per-client hash tables of sendable and receivable message types as well as the secondary receiver index hash-table were all protected by read-write locks that were acquired in exclusive mode by `RegisterAsSender()`, `RegisterAsReceiver()`, and `Disconnect()` transactions, and acquired in shared mode by other transactions. This use of read-write locks was a first attempt at achieving good concurrency, while still having fully serializable correct behavior.

Each client's queue of incoming messages is protected by a simple mutex, which is locked whenever any operation (*push*, *read*, or *delete*) is performed on the queue. To efficiently support the blocking version of the `Receive()` call, a *read* operation that sees an empty queue releases its lock on the mutex, and waits on a condition variable. A subse-

quent *push* operation that enqueues a message, which satisfies the minimum priority of the waiting *read* operation, signals the condition variable, thereby waking the reader.

Experiments showed that the overhead associated with latching can cause severe performance issues (especially when there are many actors in a highly multi-threaded environment), so a lock-free user space concurrency control mechanism was developed, which is described in the next section.

Lock-Free Data Structures To replace read-write locks for shared data structures, a hybrid multi-version concurrency control mechanism was implemented that borrows heavily from the read-copy-update (RCU) [72] paradigm, as well as rw-locks and reference-counting garbage collection. For the sake of brevity, this hybrid mechanism will be referred to as *HybridCC*. The read-write locks for every shared data structure (with the exception of per-client queues) have been replaced with instances of HybridCC.

HybridCC keeps track of a version of an object in memory, with an atomic reference count attached to it (the reference count starts at 1 for the current version of an object). There is also an atomic "current" pointer that points to the most recent version of the object. Client code that requires read-only access to the shared object loads the current version pointer, and increments the reference count for that version (thus, reading the most recent committed snapshot of the shared object). When a client is finished reading the shared object, it automatically decrements the reference count for that version. Any number of readers can access a HybridCC-managed shared object without any locking or waiting at all and observe snapshot isolation.

If client code wishes to modify a HybridCC-managed object, it first acquires a mutex (all writers are serialized), then it makes a copy of the current object version and performs arbitrary modifications on its private copy. To commit the new version, the writer atomically swaps the current-version pointer with its modified copy, then decrements the reference count of the previous version (recall that when a version becomes "current" it starts with a count of 1), and finally releases the mutex. When the atomic reference count for a version reaches zero, the version is deleted (thus preventing memory leaks for old versions). Unlike RCU, writers do not wait for all readers of an old version to finish before deleting an old version (although writers may wait for each other). This also means that HybridCC can have arbitrarily long-lived readers (readers do not have to finish within a limited "grace period"), with a version finally being deleted and memory reclaimed when the very last reader is done with it. This flexibility does, however, mean that read-

ers need to do a minimal amount of non-blocking synchronization by incrementing and decrementing reference counts, which is not required in RCU.

For each case where a read-write lock was previously used to protect a shared data structure, the latch was replaced with a HybridCC instance managing the shared object, with all code paths accessing shared data structures in the same order. For higher performance, each version's reference count and the current version pointer is placed in its own separate cache line, reducing L1 cache misses from "false sharing" when multiple cores access the same atomic words.

NUMA Optimizations Multi-socket NUMA (non-uniform memory architecture) systems present another scalability challenge for concurrency control mechanisms. HybridCC requires incrementing and decrementing an atomic reference count for every read access to a shared data structure. For a system with a single CPU socket, this isn't a problem (the counter can remain resident in the CPU's shared last-level cache). On a NUMA system, though, any modification to a shared atomic word at one CPU socket forces that shared word to be evicted from the caches of all other CPUs in the system in order to maintain cache coherency. When the other CPUs wish to access that shared word (as they are bound to do frequently) they must read remote memory, which has considerably higher latency than the socket's locally-attached memory. Benchmarks on a 4-socket NUMA server showed that total message throughput actually *decreased* by more that 50% going from 1 to 4 CPUs, even though there were 4X as many parallel CPU cores working.

To address this NUMA scaling catastrophe, the simple atomic reference counters were replaced with a two-tier NUMA aware reference counting mechanism. In the new mechanism, there is a separate private reference count for each NUMA node that resides in its own cache line (these are all initially set to 1), as well as a single global reference count whose value is initially set to the number of NUMA nodes in the system. A thread that wishes to read a HybridCC-managed data structure increments and decrements the private counter for the NUMA node that it is running on. When a writer commits a new version of an object, it decrements *all* of the per-socket counters. When a NUMA node's private counter reaches zero, the thread that decremented it then decrements the global counter, indicating that there are no longer any readers on one of the NUMA nodes. When the global counter reaches zero, the old version of the object and all of the reference counts are deleted and their memory freed. Since reads are more common than writes, the common case only modifies the local, private reference count and does not result in any remote

memory access or cache invalidations (this only happens when a new version is committed or an old version is deleted). A committed version of a HybridCC-managed object is truly read-only, so it can be freely replicated across local memory and caches for multiple NUMA nodes, resulting in fast read-only access from any CPU in the system. Experimental results in Section 4.5 show excellent scalability in a multi-socket NUMA system using these optimizations.

SQLite Transactions

SQLite [95] is a popular embedded SQL database library that supports multi-statement ACID transactions. TMB transactions are implemented as SQL queries over state stored in five tables:

- **client** - Contains a row for each connected client with columns for a unique integer ID, and timestamps for the connect and disconnect time.
- **sendable** - Contains two columns: client ID and message type ID, with one row for each message type sendable by each connected client.
- **receivable** - This table has the same schema as the **sendable** table, but for receivable message types rather then sendable.
- **message** - Contains columns for a unique serial message ID, `BLOB` for the message body, and columns for message metadata (the ID of the sender, the timestamp when the message was sent, an optional expiration time, the message priority, the type ID of the message, and a boolean flag indicating whether the message is cancellable).
- **queued_message** - Contains one row with a foreign-key reference to a row in the **message** table for each queued incoming message for each client.

In order to accelerate the performance of common queries, some indices have also been created: an index on the message type column of the **receivable** table (to allow for quickly looking up which clients are capable of receiving a certain type of message) and an index on the receiver client ID, priority, expiration time, and send time columns of the **queued_message** table (these metadata columns are denormalized copies of values from the **message** table, and this index allows quickly retrieving messages pending for a client in the proper order).

LevelDB Transaction Management

LevelDB [40] is an embedded NoSQL key-value store. It supports *put*, *get*, and *delete* operations on individual key-value pairs, as well as iterators that allow seeking and scanning over keys in order. Multiple reads can be issued against the same consistent snapshot, and multiple write operations can be combined into a single atomic batch. A minimal bus manager has been built on the snapshot isolation and atomic commit features present in LevelDB. Five different types of keys and mapped data structures are used in LevelDB that correspond closely to the five tables used in the SQLite implementation.

Zookeeper Transaction Management

Apache Zookeeper [9] is a distributed NoSQL data store with strong consistency guarantees. Zookeeper servers are usually configured as an "ensemble", with the service remaining available so long as a majority of the servers in the ensemble are up. A Zookeeper-based TMB implementation is thus suitable for clients that are running on different machines to communicate reliably across a network, with the TMB maintaining high availability thanks to the redundancy of Zookeeper servers. Zookeeper servers synchronously log data changes to disk as part of all modifying operations, and a single Zookeeper server can be used without an ensemble if high availability in a cluster is not needed.

The Zookeeper data model is a tree of nodes not unlike a conventional filesystem, except that there is no distinction between files and directories. The Zookeeper-based TMB implementation has a directory structure that is broadly similar to the ordered key structure of the LevelDB implementation, although some changes were necessary to ensure that nodes are ordered properly by lexicographical string comparison, since Zookeeper does not allow custom comparison functions to be used. The implementation of TMB transactions is also similar to the LevelDB implementation, but it was extended to handle and resolve anomalies that can arise from multiple reads in a transaction observing different versions of the data tree (unlike LevelDB, Zookeeper does not have a mechanism to provide snapshot isolation for multiple consecutive reads).

VoltDB Transactions

VoltDB [103] is a main-memory distributed SQL database that uses partitioning to improve performance. Because VoltDB and SQLite are both SQL-based relational DBMSs, the TMB implementations for each DBMS are similar, with many of the same tables and

transactions. Because VoltDB allows stored procedures to be written in Java, some TMB logic was implemented directly in the database that in other cases required implementation in the TMB client library. In order to take advantage of VoltDB's ability to efficiently execute transactions on different partitions of data in parallel, the message contents and the metadata are denormalized into the **queued_message** table, which is then partitioned on the receiver ID attribute. Stored procedures implementing the `Receive()` and the `DeleteMessages()` APIs are partitioned on client ID for parallel execution, as is a fast-path version of `Send()` for the common case of a single explicit recipient.

Persistence & Recovery Tier

In order for the TMB state to be persistent and recoverable after failures, it requires a storage component that logs transactions durably and allows consistent TMB state to be reconstructed after a failure or interruption. Each of the four third-party databases which was used to develop a TMB implementation provides durable storage and recovery of committed transactions. A custom, minimal synchronous write-ahead log was also developed (including a checksum mechanism for verifying log record integrity) that can be used to replay a TMB's history and recover its state. The native write-ahead log is very simple, using POSIX atomic I/O syscalls for writes and the `fdatasync()` syscall to synchronously flush log records when committing.

Synchronous vs. Asynchronous Logging

By default, all TMB implementations synchronously flush logs to disk when committing a transaction so that data loss is never experienced in case of a crash. Some implementations do, however, allow logging to be asynchronous so that the operating system can buffer a number of log writes together before flushing them to disk, potentially allowing both lower latency for log writes and higher overall messaging throughput, with the caveat that some of the most recent messages may be lost in the event of a crash. Asynchronous logging is optional for the native write-ahead log, as well as the LevelDB and VoltDB implementations of the TMB. By default, synchronous logging is used for the strongest possible durability, but asynchronous logging is left as an option for users that are willing to accept the trade-off for increased performance. Experiments in Section 4.5 compare both styles of logging.

Networking Tier

Finally, this section describes the networking tier, which is necessary for TMB clients running on different machines to transparently share a TMB and communicate with each other. There are two different approaches that have been evaluated in the networking tier. The first is a custom TMB network protocol that was developed on top of the GRPC cross-platform RPC framework [41]. In this protocol, there is a single TMB server that is responsible for running the TMB's transaction manager, and all client machines connect to the server. If the server crashes, clients must wait for it to become available again to continue messaging (TMB calls will time out or fail with a network error during this window, but clients can remain up and TMB state will be restored exactly as it was upon recovery). It should be noted that a number of cloud-hosting services offer hot restart for VMs that crash, quickly bringing up a replacement server connected to the same persistent disk. Although this does not give truly uninterrupted TMB service, it may give sufficiently high uptime for many applications.

The other approach to the networking tier is to leverage the built-in network transparency of an existing distributed database (in these prototypes, this is possible with both Zookeeper and VoltDB). In this case, the TMB library on clients communicates directly with Zookeeper or VoltDB servers, and transaction management is handled in the database itself. Zookeeper servers are typically configured as an "ensemble" consisting of an odd number of machines, with the overall system remaining available so long as a majority of the servers are up and the network is not partitioned. VoltDB clusters have a user-tunable "K-safety" parameter, which causes each partition of data to be replicated across K different servers in the cluster, with replicas distributed so that any K servers can fail simultaneously and the cluster will remain fully available with all partitions online. This latter approach to networking has the advantage of high availability with zero interruptions in the face of individual server failures, although it may come at performance penalty since transactions must be applied at multiple replicas instead of a single server. Operational costs are also likely to be higher in this scenario, since multiple servers with uncorrelated failure domains must be kept running. An experimental evaluation comparing both networking approaches is contained in Section 4.5.

4.5 Experiments

This section presents empirical results comparing various TMB implementations.

Stress-Test Benchmark

To evaluate the performance of different TMB implementations, a stress-test throughput benchmark was devised. This benchmark starts a configurable number of sender and receiver threads, each of which connects to an TMB instance as a client. The sender threads repeatedly send messages as quickly as possible, randomly choosing one receiver for each. The total aggregate throughput across all receivers is measured in messages per second. Experimental runs were conducted with each TMB implementation, varying the number of sender threads and measuring the impact on throughput.

Experiments were conducted to measure both the intra-node and inter-node (i.e. scale-out) scalability of the various TMB implementations. Each TMB implementation was benchmarked on a multi-socket NUMA server with four Intel Xeon E7-4850 CPUs running at 2.0 GHz (each CPU has 10 cores and 20 hardware threads with 64 GB of directly attached memory), with a four-disk striped hardware RAID as persistent storage.

For the first round of experiments, the affinity mask of the benchmark executable was set so that it would run on only one CPU socket and access only local memory.

Subsequently, another round of experiments was conducted where all four sockets were used to measure NUMA scalability. When testing the Zookeeper and VoltDB implementations, the server process was run on the same machine.

To measure inter-node performance, a cluster of dedicated virtual machine instances in the Google Compute Engine [39] cloud hosting service was configured. Servers were set up to run either the standalone TMB network protocol server or the underlying Zookeeper or VoltDB service with 8 Xeon CPU cores at 2.3 GHz, 30 GB of RAM, and a 128 GB SSD. When benchmarking the TMB network server, a single virtual server was used. For experiments with Zookeeper and VoltDB, three servers were used, meaning that the Zookeeper ensemble could tolerate the loss of one server and remain available, and VoltDB was similarly configured with a K-safety factor of 1. A number of "application" nodes were brought up to run the benchmark with 4 Xeon CPU cores and 15 GB of RAM. Experimental runs were conducted with 1, 2, 4, 8, and 16 application nodes, varying the number of threads running on each.

Additional experiments compare the performance of the TMB against existing best-of-breed message-oriented middleware, specifically Apache ActiveMQ (a message broker for the Java Message Service) and the Spread Toolkit (a multicast distributed messaging service with virtual synchrony). The stress-test benchmark was ported to ActiveMQ and Spread, and it was run under the exact same cloud server configuration that was used to

evaluate distributed TMB implementations (three 8-core servers running ActiveMQ brokers or Spread daemons, 1-16 quad-core application nodes running client threads). ActiveMQ was configured to use replicated LevelDB storage for persistent queues, with the three servers acting as a quorum with automatic failover for high availability. The three Spread daemons were configured as a single "segment" with safe, and fully atomic, multicast. Note that Spread does *not* support durable message queues (i.e. messages can be lost if daemons fail), so ActiveMQ and TMB are somewhat disadvantaged by their requirement to log messages durably in this pure performance comparison.

Distributed Search Application

A sample distributed search application was also developed using the TMB. The structure of this sample application follows the example presented throughout Section 4.3, with the addition of a simple term frequency-inverse document frequency (TF-IDF) ranking function. A client submits a set of keywords for text search, and servers scan plain text documents and return a hit list with frequency counts for each keyword in each matching document. The client then counts the total number of matching documents for each keyword to determine each keyword's inverse document frequency, and finally ranks all the matching documents according to TF-IDF.

Four search servers (matching the application node configuration above) were run, each containing partitions of a large English plain text corpus with 44 partitions in total, each approximately 100 MB in size. Each data partition was replicated on two different servers, and the replicas were distributed so that each server contained a copy of 22 different partitions, and no two servers had the same set of replicas. A client submits a keyword search request to one server for each partition.

A straggler node was then simulated by causing one of the servers to make four passes over the same data before responding.

The client was set up to submit tied messages to both servers for each partition in an attempt to mitigate the performance impact of the straggler (servers cancel a request for their peer when they begin working).

The VoltDB-based TMB implementation was used for this experiment. Except where otherwise noted, all results below use the stress-test benchmark.

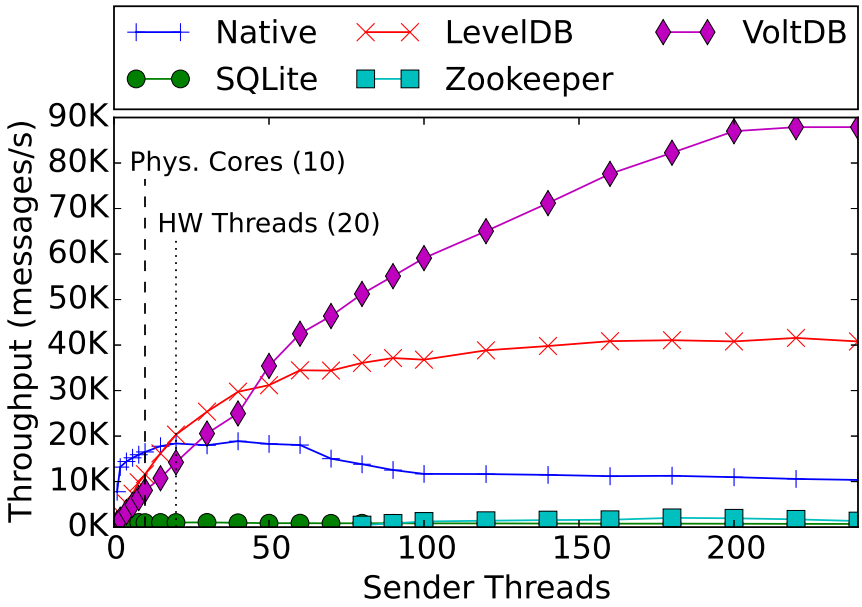


Figure 4.2: Single Socket TMB Performance (Strong Durability)

Single-Socket Performance

Figure 4.2 shows the relative performance of five persistent and durable TMB implementations when running on a single CPU socket (10 cores / 20 hyperthreads). Recall that these implementations were described in Sections 4.4 through 4.4. In Figure 4.2, the label "Native" indicates the use of the TMB in-memory transaction manager in combination with the TMB write-ahead log. The native log, LevelDB, and VoltDB are all configured to use synchronous logging for the strongest possible durability.

The first important finding from this experiment is that the Native, LevelDB, and VoltDB implementations vastly outperform and out-scale the SQLite and Zookeeper implementations. Both LevelDB and VoltDB scale throughput well with additional threads before eventually leveling out when the number of sender threads far exceeds the number of hardware threads. The Native implementation is I/O-bound and has flatter performance, but it should be noted that it achieves the highest throughput when the number of threads is equal to the number of physical CPU cores (10), and achieves throughput only 9.5% below LevelDB's peak of 20318 messages/s when the number of threads is equal to the number of hardware threads (20). VoltDB ultimately scales up to a higher peak throughput when threads are heavily oversubscribed to cores.

Figure 4.3 shows the performance impact of optional asynchronous logging. With-

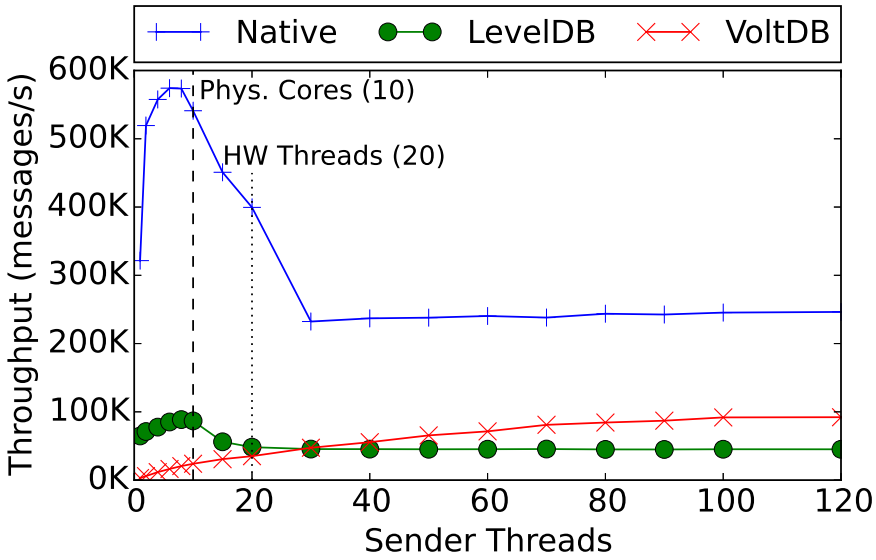


Figure 4.3: Single Socket TMB Performance (Asynchronous Logging)

out forcing `fdatasync()` syscalls to to flush logs to disk, all three of the persistent TMB options that support asynchronous logging achieved higher messaging throughput. The effect was especially pronounced for the Native TMB, which achieves peak throughput more than 5X higher than any other TMB implementation when the number of sender threads is equal to the number of CPU cores (10). Oversubscribing threads to CPU cores caused throughput to be reduced from peak for both the Native and LevelDB TMB implementations as more threads contended with each other in the I/O and logging subsystem. Throughput at 120 threads was only 43% as high as the peak at 8 threads for Native, and 51% as high as the peak at 8 threads for LevelDB. VoltDB, on the other hand, did not experience a drop-off due to contention between many threads (this may be because VoltDB's lightweight "command logs" are partitioned and transactions executing serially within a partition never interfere with each other and can buffer log records freely).

Figure 4.4 shows the performance impact of using the TMB in-memory transaction manager as an in-memory cache as described in Section 4.4, using LevelDB as the underlying storage engine as an instructive example. Using the custom TMB transaction manager as an in-memory cache improves the scalability of the LevelDB-based TMB considerably, regardless of whether synchronous logging is used. In the asynchronous case, the TMB cache eliminates the need for any explicit snapshotting as well as any read con-

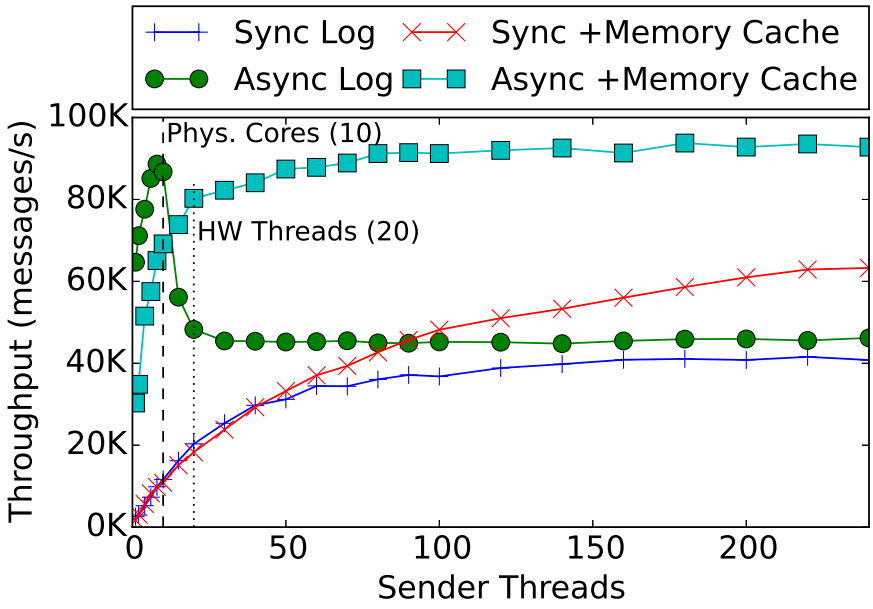


Figure 4.4: Effect of Memory Cache (LevelDB - 1 Socket)

tention, reducing the interaction with LevelDB to a series of small point writes, with average throughput of up to 93764 messages/s with 180 sender threads. In the synchronous case, the TMB cache allows the message bus to continue scaling up to 63272 messages/s with 240 sender threads, a 55% improvement over sync logging without the cache.

The TMB in-memory cache was also tested with the SQLite, Zookeeper, and VoltDB-based TMBs. The cache resulted in only a slight improvement for SQLite (SQLite calls would still lock to serialize transactions), and had little impact on VoltDB (this is to be expected, since VoltDB is already a main-memory database). The Zookeeper implementation experienced a 3.3X improvement in peak throughput with the cache, although the highest average throughput achieved was still only 6685 messages/s.

In-Memory Concurrency Control

The in-memory TMB transaction manager was also benchmarked without any persistent storage to investigate whether there were any performance or scalability issues in this core bus-management component. In particular, different concurrency control mechanisms were compared: conventional rw-locks vs. the HybridCC mechanism which is completely lock-free for readers. These experiments showed that using rw-locks does not scale, with overall throughput "flatlining" at about 1.5 million messages/s. HybridCC, on the other

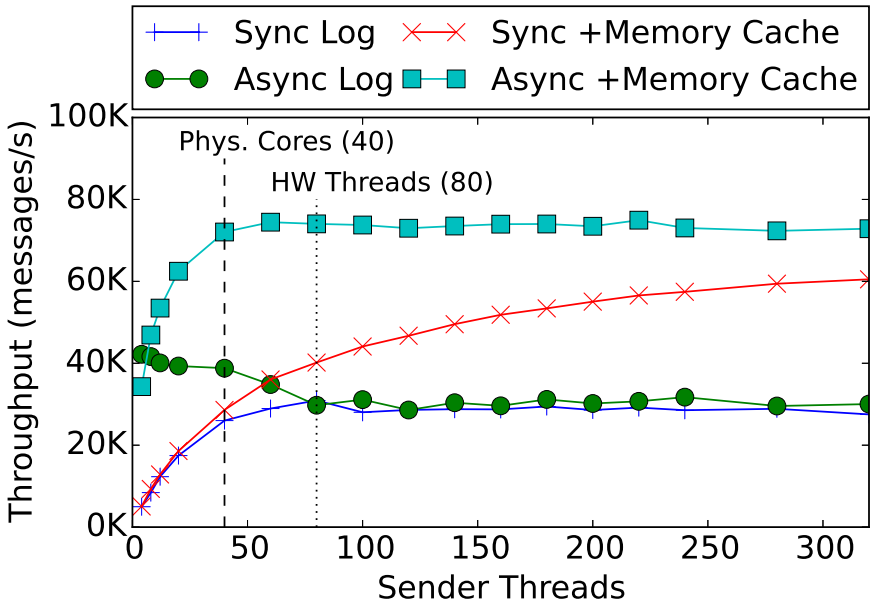


Figure 4.5: 4 Socket NUMA Performance (LevelDB Storage)

hand scales nicely, reaching a peak of 6.9 million messages/s with 160 sender threads. Note that, even without the benefit of HybridCC, the "pure memory" TMB implementation substantially outperforms the best persistent implementations, since DRAM access is much faster than disk access.

NUMA Scale-Up

This experiment moves from using a single CPU to using all four CPUs in the test server. Since accessing remote memory in a NUMA system can be considerably slower than local memory, three sets of tests were run: one where threads were not affinitized at all (i.e. free to run anywhere), one where each thread was affinitized to run on one particular CPU socket, and one where threads were affinitized to CPU sockets *and* would only send messages to their peers on the same socket.

Figure 4.5 shows the performance of the various flavors of the LevelDB-based TMB implementation under NUMA. For this experiment, the LevelDB-based TMB implementation was used, as it achieves the best performance of the persistent and synchronous TMB implementations for NUMA. The results shown in Figure 4.5 are for affinitized threads that send messages to any other threads on any socket, although the other result results showed almost no difference in throughput for the other affinity configurations (this was

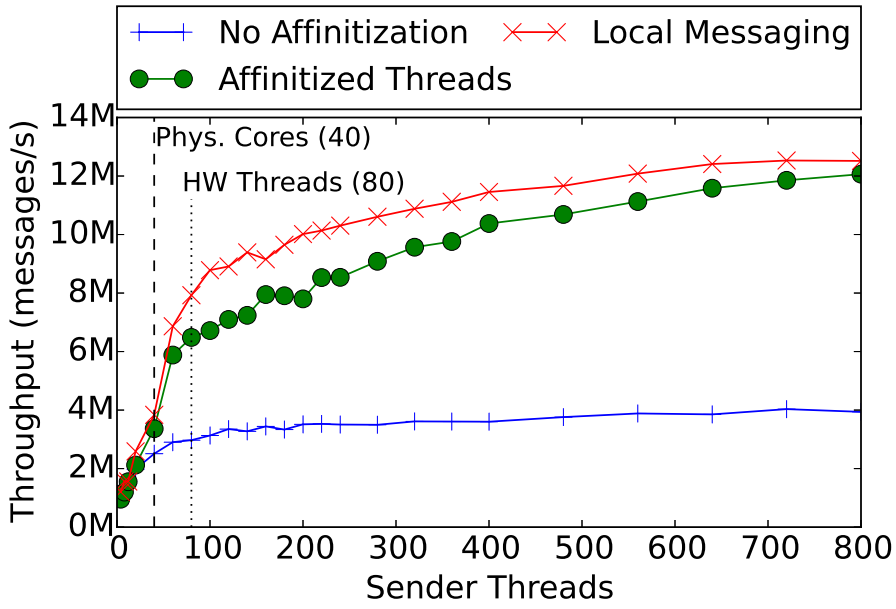


Figure 4.6: 4 Socket In-Memory NUMA Performance - Thread Affinity

true for the other persistent TMB implementations as well, suggesting that remote memory access is not a significant bottleneck compared to disk I/O). Without the TMB in-memory transaction manager as a cache, throughput using both synchronous and asynchronous logging converges to about 30000 messages/s. With the cache, throughput scalability improves, peaking at about 75000 messages/s (a reduction of 20% compared to the single socket) with asynchronous logging and 60500 messages/s with synchronous logging (nearly the same as the single socket).

The same 4 socket NUMA scale-up experiments were also on the other TMB implementations. The SQLite and the Zookeeper implementations continued to perform poorly with NUMA. VoltDB also suffered considerable performance degradation compared to the single node case, never exceeding 20000 messages/s for any experiment, even with affinitized threads. Although client threads were affinitized to individual CPU nodes, their corresponding VoltDB partitions were not, meaning that TMB operations would frequently access remote memory, which is symptomatic of the fact that VoltDB is not particularly optimized for intra-node scalability on NUMA, instead focusing on scaling across clusters.

The NUMA scalability of the TMB in-memory transaction manager was also tested without any persistent storage (i.e. a "pure memory" TMB) with affinitized threads that

communicate with their peers on the same socket. With the original NUMA-oblivious HybridCC implementation, scalability is poor, and throughput flatlines at about 2.3 million messages/s (only 1/3 the throughput of the single-socket case). Switching to the modified HybridCC implementation with two-tier NUMA-aware reference counters solved this problem, eliminating a great deal of the writes to shared memory (and hence remote cache invalidation and memory access), and allowing the TMB transaction manager to scale well in a NUMA environment, achieving over 12 million messages/s throughput. Figure 4.6 compares the pure memory TMB throughput when running under NUMA for unaffinitized threads, threads affinitized to run on a particular CPU socket, and threads affinitized to run on a single CPU socket that also only send messages to their peers on the same socket. As one would expect for a NUMA application, affinitizing threads is necessary to achieve good performance scaling. Only messaging peers on the same socket improves throughput further. Interestingly, affinitized threads that communicate globally with all their peers do not perform dramatically worse than the local messaging-only case, which suggests that any communication pattern using the TMB scales well in a NUMA environment, so long as threads are affinitized. The explanation for this behavior is that a *push* operation on another client's queue already incurs the overhead of locking a mutex, and writing to remote memory is limited and localized, unlike the pathological case of NUMA-oblivious shared counters.

Cluster Scale-Out

Figure 4.7 shows the results of experiments using the TMB network protocol server for communication in a cluster. The server uses the TMB in-memory transaction manager, and uses LevelDB (with synchronous logging) for persistent storage [2]. 1, 2, 4, 8, or 16 application nodes were connected to the TMB, and throughput was measured with the stress-test benchmark. This graph shows a similar relationship between the number of sender threads and the message bus throughput regardless of the number of application nodes (the line for 2 nodes is omitted for clarity; it is virtually identical to the 4 node case). This shows that messaging throughput is effectively network-agnostic, with the TMB delivering the same throughput for a wide range of cluster sizes. The data for 1 node and 16 nodes show throughput that is slightly diminished relative to the other cluster

[2]LevelDB was chosen for the TMB Net Server's persistent storage, as it showed the best performance and scalability characteristics of the (synchronous) persistence-tier options in the single-node experiments described previously.

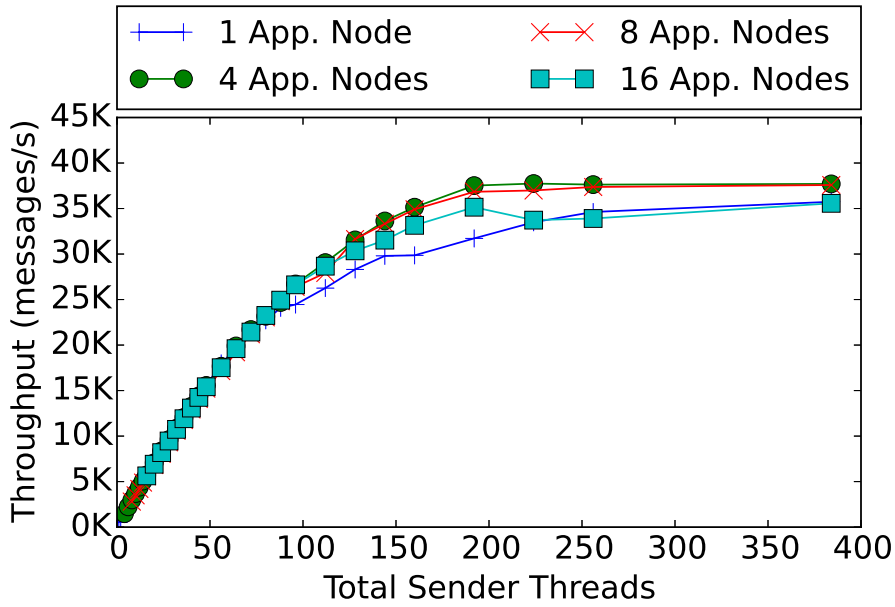


Figure 4.7: Cluster Scale Out (TMB Net Server with LevelDB Storage)

sizes. The 1 node case is limited by the load from heavily oversubscribing threads to CPU cores on the application node, while the 16 node case is limited by high CPU load and many open network connections on the server node.

Figure 4.8 shows the results of a similar experiment for the distributed VoltDB TMB implementation. Recall from Section 4.4 that a distributed database like VoltDB allows for a different approach to networking, with clients communicating directly with VoltDB servers and transaction management handled in VoltDB itself. VoltDB was run on three server nodes (a cluster with K-safety factor of one), then application nodes were connected to the TMB and ran the stress test benchmark. Once again, despite the different approach to the networking tier, messaging throughput scales in an effectively network-agnostic fashion, with throughput slightly diminished when client nodes are heavily loaded (the 1 application node case) or server nodes are heavily loaded (the 16 application node case).

Experiments were also conducted with a variable number of VoltDB server nodes (while fixing the number of application nodes at 8). Adding additional servers increases the average message throughput in the cluster for any number of sender threads, with the difference more pronounced for a higher number of threads. It should be noted, though, that the throughput scale-up from adding more VoltDB servers is less than linear (for instance, with 384 sender threads, a configuration of 6 VoltDB servers achieved 27% higher

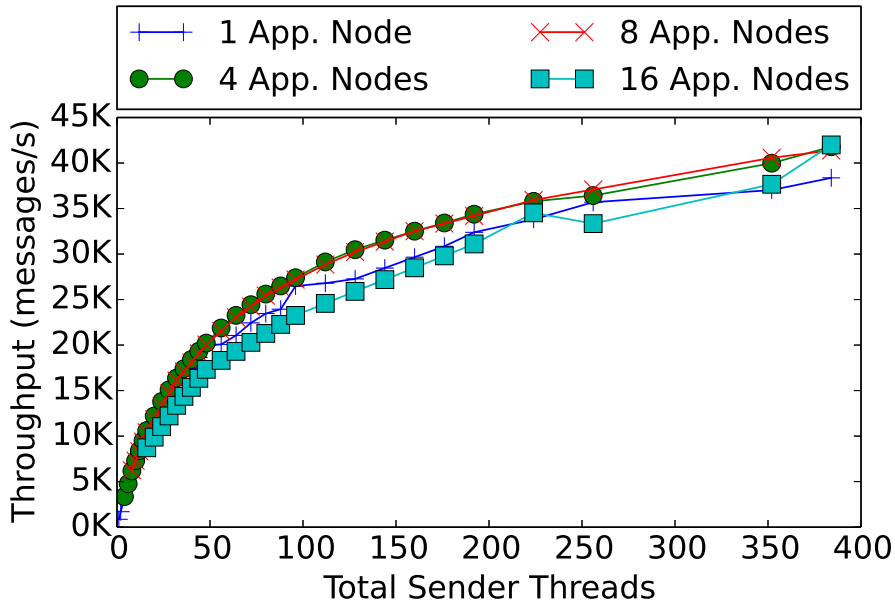


Figure 4.8: Cluster Scale Out (TMB on VoltDB)

throughput than 3 servers, while 9 servers achieved 42% higher throughput).

Finally, the Zookeeper TMB implementation was evaluated in the cluster environment. Similar to the VoltDB implementation, TMB on Zookeeper leverages Zookeeper's own network protocol to have TMB clients connect directly to Zookeeper servers in an ensemble. As with the single-node case, performance was underwhelming compared to the alternatives. The highest throughput was 4783 messages/s for 8 application nodes. Unlike the TMB network server and TMB on VoltDB, the relationship between sender threads and throughput was not cluster-agnostic, with smaller clusters of application nodes experiencing significantly lower throughput than larger clusters.

Comparison With ActiveMQ & Spread

Figure 4.9 shows the message throughput of the stress-test benchmark using the TMB network server backed by LevelDB storage and TMB on VoltDB vs. the Apache ActiveMQ message broker and the Spread Toolkit. With 8 application nodes, ActiveMQ's message throughput is in the range of 950 to 990 messages/s regardless of the number of sender threads. Spread achieves its highest throughput when there are few threads running on each node (30205 message/s with only a single sender and receiver thread on each VM), with performance diminishing due to contention as additional threads are added. On the

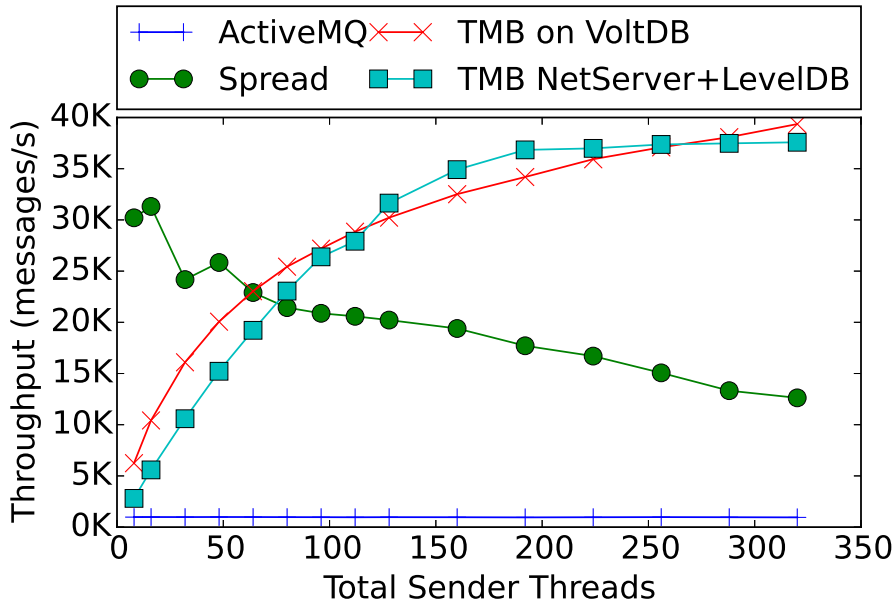


Figure 4.9: ActiveMQ vs. Spread vs. TMB (8 App Nodes Cluster)

other hand, the two TMB implementations' throughput scales with additional threads, with throughput from 6.4X to 34X higher than ActiveMQ depending on the number of threads. Either TMB implementation also achieves higher throughput than Spread above 64 sender threads (the number of CPU cores in the cluster) and, unlike Spread, the TMB stores messages durably.

Note that both the TMB network protocol server and TMB on VoltDB have similar throughput curves. The comparison is not entirely fair, however, as the TMB network server uses only a single machine, while TMB on VoltDB uses three. On the one hand, this means that operational costs for the TMB network server should be lower. On the other hand, TMB on VoltDB is more resilient to server failures, and can maintain high availability with zero downtime if one server fails.

These results show that the TMB design approach, which leverages either a custom-built lightweight transaction manager and network protocol or a high-performance distributed DBMS, compares favorably with leading purpose-built MOM systems.

Distributed Search Results

The sample text search and ranking application was run using a cluster of four search servers each containing replicas of a large partitioned text data set as described at the

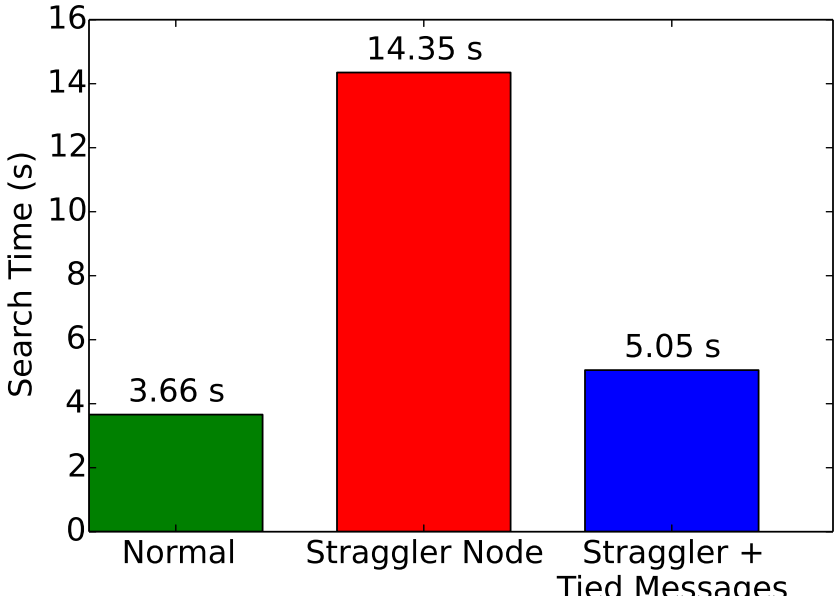


Figure 4.10: Search Latency In Distributed Search Application

beginning of this section. The results for this experiment are shown in Figure 4.10. A full-text keyword search on the unloaded servers completed in 3.66s. A straggler node was then simulated, which caused the completion time to increase to 14.35 s. Finally, cancellable tied messages were enabled, which allowed the search to complete in 5.05 s despite the slow performance of the straggler node. This demonstrates the effectiveness of tied messages in dealing with lagging and unreliable components in a distributed system.

4.6 Conclusion

This chapter has presented the Transactional Message Bus, a new communication framework that provides a superset of desirable features from existing message-oriented middleware systems and is built as a modular, reusable software library. The development of the TMB was originally motivated by the need for a communication fabric between the scheduler and workers in Quickstep that would scale well in multi-socket NUMA servers, and would also provide a foundation for a future distributed version of Quickstep.

A survey of message-oriented middleware (MOM) revealed many features that are desirable in a messaging system, including features relating to reliability (e.g. at-least once delivery, persistent queues, high availability), consistency (e.g. virtual synchrony),

and manageability (e.g. tied messages, priority and deadlines). Crucially, no preexisting MOM system delivered the complete set of these features (see Table 4.1). The first major contribution of the TMB was the detailed definition of the logical semantics and client interface for a message bus incorporating all of these features.

Building on the foundation of well-defined semantics for messaging, the next contribution of the TMB work was the development of practical TMB implementations. The TMB implementations expand on Gray's insight that queues are databases [44] and treat all aspects of message bus management as a form of transaction processing (hence the term *Transactional* in Transcational Message Bus). Existing transaction-processing systems, including both relational databases (SQLite, VoltDB) and NoSQL data stores (LevelDB, Zookeeper) can be leveraged to implement the full TMB semantics. A "native" written-from-scratch TMB stack was also developed, consisting of a custom in-memory transaction manager, networking protocol, and write-ahead logging mechanism for persistence and recovery.

Finally, an experimental evaluation of the various alternative TMB implementations was carried out with a stress-test benchmark that measures overall messaging throughput with an increasing number of clients connected to the bus. This study revealed a number of interesting and at times surprising results. Firstly, it showed that both SQLite and Zookeeper are a poor basis for a messaging bus, with throughput much lower than the alternatives. Secondly, it revealed scalability problems with widely-used in-memory concurrency control primitives like rw-locks and motivated the development of a much more scalable hybrid concurrency-control mechanism combining ideas from rw-locks and the read-copy-update paradigm (this hybrid mechanism was further tuned for high performance in a multi-socket NUMA setting). In the NUMA setting, it was found that, so long as the threads using the TMB are themselves affinitized to particular sockets, the performance penalty for inter-socket messaging is surprisingly modest, and an application need not be concerned about the "distance" between peers communicating using the TMB. In benchmarking distributed TMB implementations (both a native TMB network server and a TMB running on a VoltDB cluster), a property of network-agnostic scalability was revealed; the relationship between overall messaging throughput and the number of client threads was similar regardless of whether those client threads were distributed over 1 to 16 nodes. Finally, the performance of the distributed TMB implementations was compared with other popular messaging frameworks (ActiveMQ and Spread) and the TMB was able to achieve throughput far higher than ActiveMQ and competitive with Spread.

The in-memory version of TMB satisfies the original need for a high-performance messaging system within a single-process version of Quickstep that can scale up for many threads across multiple NUMA sockets. The distributed versions of the TMB also provide a path forward to develop a distributed version of Quickstep that uses exactly the same interfaces and semantics to communicate over the network.

5 EXPERIMENTAL EVALUATION OF THE QUICKSTEP DATABASE MANAGEMENT SYSTEM

5.1 Introduction

This chapter presents results from a series of empirical experiments on end-to-end query processing in the Quickstep database management system, whose overall architecture was described in Chapter 2. These experiments complement the focused evaluation of the Quickstep storage engine (see Section 3.4) and the TMB communications framework (see Section 4.5). Note the key goal of the empirical evaluation is to highlight the advantages of various aspects of the Quickstep design, rather than an extensive evaluation with various other systems. The target for Quickstep is single node system with large main memory and key aspects of such comparisons have been presented before (e.g. previous publications on WideTable denormalization techniques [71]). Thus, the focus here is largely on evaluation of the different aspects of designs choices made in Quickstep.

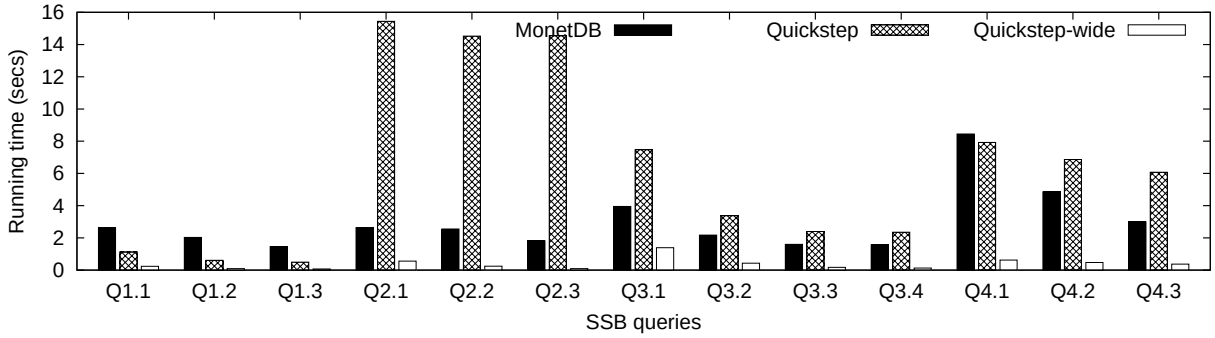
Experimental Setup

For these experiments, a server with four Intel Xeon E7-4850 CPUs clocked at 2.0 GHz was used. Each CPU has 10 cores and 20 hyperthreading hardware threads. The machine runs CentOS Linux 7.0 with Linux Kernel 3.10.0. The server has four NUMA nodes, one for each socket, and 64 GB of directly-attached memory per NUMA node (total 256 GB for the machine).

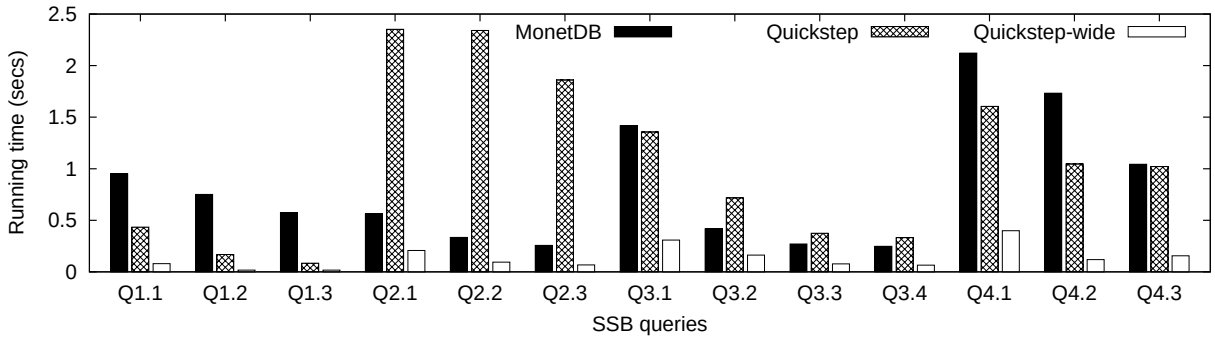
5.2 Comparison to MonetDB

This section presents results comparing Quickstep to a leading open-source in-memory analytical DBMS – MonetDB (version 11.19, released in October 2014). MonetDB is a full-fledged column-oriented DBMS developed at CWI. It is designed to provide high performance on complex analytics queries, and is optimized for modern multi-core CPUs.

This evaluation uses the Star Schema Benchmark (SSB) [78]. The SSB is based on the TPC-H benchmark, but is designed to measure performance of database systems in support of classical data warehousing applications. The goal here is to present end-to-end results that run through the system. The front-end parser and optimizer in Quickstep are,



(a) Single-thread execution



(b) Multi-thread execution

Figure 5.1: MonetDB vs. Quickstep (SSB Benchmark - scale factor 10).

admittedly, not yet ready to run complex workloads like TPC-DS and TPC-H, but development is ongoing to address those limitations. It should be noted that the SSB has been extensively used in previous work [2, 99], and it helps illustrate the base performance of Quickstep on this admittedly simple class of analytic queries.

Since MonetDB is not tuned for NUMA systems, this comparison was restricted to just a single socket. SSB datasets at scale factor 1 and 10 were used. The total sizes of the raw data set are approximately 0.6GB and 6GB, respectively.

In the evaluation below, each query in the SSB was run 10 times. The average execution time for the 10 runs for each query is reported. Both MonetDB and Quickstep were warmed up before each experiment. The server thread(s) were also pinned on particular CPU core(s), so that no thread migration occurs during this experiment.

Figure 5.1 shows the run times of MonetDB and two Quickstep variants with the 13 queries in the SSB benchmark. In the figure, the tag *Quickstep* refers to the basic settings: a column-store was used, all variable-length attributes and all filter attributes were compressed (i.e. the attributes that are used in selection predicates), and BitWeaving/V in-

dices were built on all filter attributes. Below, the tag *Quickstep-wide* refers to the method with the WideTable technique: the SSB schema was flattened out, and all tables were pre-joined together to form a single denormalized table; all attributes in the wide table were compressed and BitWeaving/V indices were created on all filter attributes. Experiments were also conducted on other variants of Quickstep. The experimental results for other Quickstep variants are shown in Section 5.3.

Single thread performance comparison

As shown in Figure 5.1a, with a single-thread execution, Quickstep is over 2X faster than MonetDB on a subset of the SSB queries, e.g. Q1.1, Q1.2, and Q1.3. These queries include joins with the `Date` table that contains only about 2500 tuples. Since the hash table of the `Date` table can entirely fit in CPU cache, the join operator takes only a small portion of the overall execution time, while the scan operators are the bottleneck for these queries. Quickstep performs these scan operations efficiently thanks to the use of BitWeaving/V indices, and significantly reduces the query execution times for these queries.

On the other hand, Quickstep is, on average, 2.2X slower than MonetDB on other queries. The performance gap is further increased for the queries Q2.1, Q2.2, and Q2.3, which contain joins with two large dimension tables. Quickstep performs poorly on the Q2 variants mainly because the join operator in Quickstep is not yet optimized for the CPU cache performance – a single global hash table is used, and it can grow considerably larger than the CPU cache size. As a result, the number of cache misses quickly increases and hinders overall performance. (Part of future work is to overcome these limitations and employ techniques that optimize the join operator for CPU cache efficiency.)

As can be seen in Figure 5.1a, Quickstep-wide outperforms MonetDB on all queries, with over 10X in speedup for a majority of the SSB queries. Quickstep-wide takes advantage of the denormalization method, which evaluates complex join queries using sequential scans over compressed values with the BitWeaving technique. For a small subset of SSB queries, the speedup of Quickstep-wide over MonetDB is relatively reduced. For example, Quickstep-wide is only 2.8X faster than MonetDB on the query Q3.1. The reduced speedup is mainly due to the fact that the group-by operator makes up a large portion of the total run time for these queries. The group-by operator uses a hash table on the group-by attributes, which does not fit in CPU cache and hinders the overall performance for these queries (this is another manifestation of the same problem with overly-large hash tables that was encountered with join queries on vanilla Quickstep without WideTable).

Multithreading performance comparison

In the next experiment, the number of threads was set to 40, which is equal to the number of processors (cores). As noted earlier, since the execution engine of MonetDB is not NUMA-aware, Quickstep was not forced to allocate memory space on all NUMA nodes or to use NUMA-aware partitioning. Instead, in order to make a fair comparison with MonetDB, the OS was relied upon to manage the memory allocation and decide the physical memory location.

Figure 5.1b shows the performance comparison between Quickstep and MonetDB with multithreading. Unlike the single thread execution, Quickstep outperforms MonetDB or shows comparable performance on most all queries, except for Q2.1, Q2.2, and Q2.3 (which still suffer from a hash-join implementation that is not yet cache-optimized). This improvement demonstrates the utility of Quickstep's block-based storage architecture and workflow-based scheduler, which naturally make use of the parallelism in many stateless worker threads.

Not surprisingly, Quickstep-wide also outperforms MonetDB when running with 40 threads, with an average 6.7X speedup across all queries in the SSB benchmark. However, the relative speedup of Quickstep-wide over MonetDB generally decreases with multithreading, compared to a single-thread execution. In the 40 thread case, the worker threads in Quickstep-wide sequentially access memory at a high speed, and quickly reach the maximum memory bandwidth in the system. Since the OS allocates memory from a single NUMA node, the memory channel between the NUMA socket and the associated memory bank becomes the bottleneck to transfer all data into all cores. The bandwidth of the memory channel is only about 10GB/s in the system, which significantly limits the scalability (to multi-cores) of Quickstep-wide. It is expected that Quickstep-wide will likely see better scalability by either running on a newer architecture (e.g. the maximum memory bandwidth of the Intel Xeon E7 v3 families is 102GB/s [55]) or allocating memory across all NUMA nodes and fully optimizing even complex queries for NUMA (see preliminary results in Section 5.4 that show promising early gains from NUMA-aware data placement and execution for simple queries).

5.3 Quickstep variants

In this section, the flexible block-based storage manager in Quickstep is used to produce seven Quickstep variants. The space requirements and the query processing performance

of these variants is then analyzed. The evaluation below focuses on four aspects of the Quickstep engine: storage format, compression, indexing, and denormalization (building on the experiments presented in Section 3.4, which contains in-depth experimental results regarding the first three of these aspects for simple scan queries). To allow for comparison, the same settings are used as in the previous experiment (cf. Section 5.2), and the SSB benchmark with scale factors 1 and 10 is again used.

The seven Quickstep variants that were produced are *col*, *col-zip*, *col-bw*, *wide-row*, *wide-col*, *wide-col-zip*, and *wide-col-bw*. Here, the tag *col* and *row* refer to column-store and row-store formats, respectively. The tag *wide* refers to a denormalized method, where the SSB schema is flattened out and all tables are pre-joined to form a single WideTable, with the original queries on normalized tables translated into equivalent queries on the WideTable. The tag *zip* indicates that all attributes in the storage layout are compressed (only variable-length attributes and filter attributes are compressed for normalized tables). Quickstep automatically chooses the compression method between ordered dictionary compression and truncation based compression (for integers), based on the compression ratio and query performance. Finally, the tag *bw* means that all filter attributes have BitWeaving indices built on them. Note that the tag *bw* implicitly indicates the *zip* tag.

Quickstep variants	storage format	compression	indexing	denormalization	space (SF10)
col	column	No	None	No	4.9 GB
col-zip	column	Yes	None	No	4.5 GB
col-bw	column	Yes	BitWeaving	No	4.6 GB
wide-row	row	No	None	Yes	30.2 GB
wide-col	column	No	None	Yes	29.6 GB
wide-col-zip	column	Yes	None	Yes	7.0 GB
wide-col-bw	column	Yes	BitWeaving	Yes	7.6 GB

Table 5.1: Characteristics of the seven Quickstep variants.

Table 5.1 summarizes the characteristics and the space requirements of the seven Quickstep variants with the SSB benchmark. For scale factor 10, the total size of the raw data set is approximately 6GB. The denormalization technique dramatically increases the database size from 4.9GB to 30.2GB (or 29.6GB for the columnar storage format). Compression is effective in mitigating this, however, reducing the size of the denormalized table to 6.0GB,

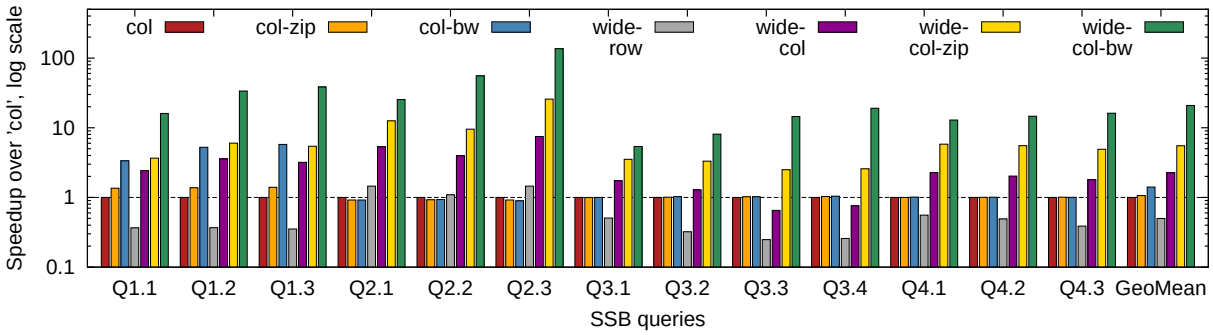


Figure 5.2: Single-thread performance comparison across Quickstep variants

which takes 40% - 55% more space than the normalized variants. The BitWeaving indices need an additional 0.6GB and 0.1GB of space for denormalized and normalized variants, respectively.

Figure 5.2 shows the performance of the seven Quickstep variants with the single-thread execution of the 13 queries in the SSB (scale factor 10). The performance is normalized with respect to that of the *col* variant. (Note that the y-axis has a log scale.)

As can be seen in Figure 5.2, the compression technique and the BitWeaving technique improve the performance of Quickstep for a subset of queries, i.e. Q1.1, Q1.2, and Q1.3. The join operators in these queries create a hash table on the `Date` table that can entirely fit in CPU cache, and consequently takes only a small portion of the total execution time. As a result, with the use of the compression and BitWeaving techniques, the *col-zip* and *col-bw* variants significantly speed up the scan operators, and thus outperform the *col* variant on these queries. However, these techniques have negligible performance impact on the other queries, as the join operators are the bottleneck in these other queries.

Surprisingly, with the use of the simple denormalization technique, the *wide-row* variant is actually slower than the *col* variant for nearly all the SSB queries. The reasons are threefold. First, although the *wide-row* variant might simplify query processing by converting join operators into (potentially faster) scan operators, it can slow down the access to each individual tuple. Access to individual tuples is slower because a tuple in the denormalized table is generally much larger than the tuple(s) in the original normalized tables, as there are more attributes in the denormalized table. Consequently, memory bandwidth is wasted in fetching attributes that are not needed for query processing. Second, as the denormalized table is typically much larger than the original tables, the *wide-row* variant requires a larger memory footprint and may reduce the locality and efficiency of the CPU cache. Third, with the use of the denormalization technique, the scan operator may need

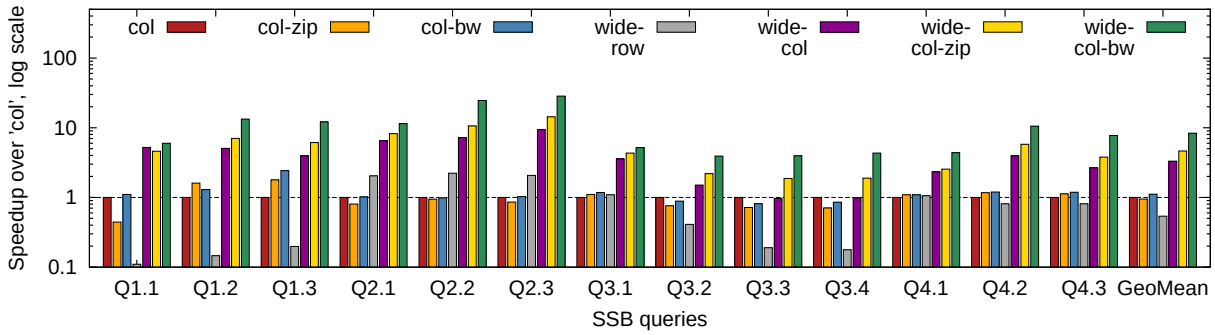


Figure 5.3: Multithreaded performance comparison across Quickstep variants

to access more tuples to evaluate the selection predicates on the original dimension table.

Nevertheless, these limitations can be overcome by combining the columnar storage layout (*wide-col*), compression (*wide-col-zip*), and BitWeaving/V indices (*wide-col-bw*) with the simple denormalization technique. As shown in Figure 5.2, by storing all blocks of the demoralized table in a columnar storage format, the *wide-col* variant outperforms the *col* variant for all the SSB queries. These results are consistent with previous studies [2, 71]. The *wide-col-zip* variant compresses all attributes in the denormalized table, achieving an additional 2X speedup over the *wide-col* variant. Finally, with the use of the BitWeaving scans, the *wide-col-bw* variant is up to 10X faster than the *wide-col-zip* variant. Collectively, the fastest variant, *wide-col-bw*, is on average 20.9X and 41.8X faster than the *col* and *wide-row* variants, respectively, across the 13 SSB queries.

Figure 5.3 illustrates the performance of the seven Quickstep variants when running with 40 threads (at scale factor 10). Compared to the single-thread execution (see Figure 5.2), the speedups of the *wide-col-zip* and *wide-col-bw* are relatively reduced. The scan operators in these two variants access data at a higher speed than other variants, and saturate the maximum memory bandwidth of the system, causing the relative speedup to be reduced. Note that for this experiment, memory was from a single NUMA node. The memory accesses are bottlenecked on the the memory channel between the CPU and the memory bank in the NUMA node. NUMA-aware data partitioning and work scheduling can help to balance the load on memory channels in the system, helping to alleviate this issue (this is demonstrated by preliminary experiments on NUMA-aware data placement in Section 5.4).

5.4 Additional Experiments

Some additional experiments were carried out to evaluate NUMA-aware data partitioning and mid-query elasticity in Quickstep. These experiments were conducted with a TPC-H data set at scale-factor 10. Although the experiments described in this section are somewhat preliminary and are not discussed in great detail, they do provide a a promising early look at the effectiveness of Quickstep features that are currently under heavy development.

NUMA Awareness

A natural join between the TPC-H *lineitem* and *orders* tables was carried out. For the non-NUMA version of the experiment, buffer pool memory for blocks was simply allocated by `mmap`, with the OS kernel deciding where to physically place pages, and a single global hash table was used for the join. For the NUMA-aware version of the experiment, the two relations were both hash-partitioned on the join key attributes, with each partition stored in local memory on a particular NUMA node. Similarly, in NUMA-aware join execution, a separate hash table was constructed for each pair of matching partitions, with the hash table's memory allocated from local memory on the same NUMA node. In both cases, the test query was run using 1, 2, or 4 sockets of the test system (using all 20, 40, or 80 hardware threads, respectively). In the NUMA-aware case, the foreman was configured to send only NUMA-local work orders to worker threads pinned on a particular socket.

Going from 1 to 2 sockets, one would hope to see a perfectly linear 2X speedup in execution time. Without NUMA-awareness, the speedup is only 1.47X. On the other hand, with NUMA-aware data partitioning and work scheduling, the speedup is a nearly-linear 1.91X. Going to 4 sockets, execution time is actually 9% *worse* for the non-NUMA-aware case with 4 nodes than it is with 2, despite using twice as many CPU cores. NUMA-aware execution continued to improve with 4 sockets, although the speedup was a more modest 1.23X over the 2 node case, or 2.35X relative to the single node. These results provide some early validation for ongoing work on developing NUMA-aware query execution techniques, and development efforts continue to fine-tune NUMA-awareness techniques in hopes of approaching linear performance scaling with 4 or more NUMA nodes.

Instant Elasticity

Another set of preliminary experiments was carried out to evaluate "instant elasticity" in Quickstep, i.e. whether additional workers can be added to an in-progress query to make it finish faster. For this experiment, a natural join on the TPC-H *lineitem* and *orders* tables was again carried out, although in this case all worker threads were pinned to a single NUMA socket to avoid confounding NUMA effects in the experiment. The total number of work orders needed to execute the query was determined up front, and the rate at which work orders were completed was measured. The query started execution with a single worker thread, and at each quartile boundary of query completion (i.e. 25% of work orders completed, then 50%, then 75%) the number of worker threads was doubled (increasing to 2, then 4, then 8 workers). As soon as new workers become available, the Foreman begins scheduling work for them. The rate of work order completion scaled almost perfectly linearly as more workers were added in this experiment, demonstrating that instant elasticity is indeed possible in Quickstep. It should be noted, however, that there are some queries where workers scan data from main memory so quickly that they saturate the available memory bandwidth to a CPU socket (e.g. the WideTable scans with 40 threads illustrated in Figure 5.3), so perfect linear scaling will not always be possible.

6 CONCLUSION

This dissertation has described the design and implementation of the Quickstep database management system, which has been underway since 2011 at the University of Wisconsin. The Quickstep project was started to investigate how in-memory analytic data processing platforms might be redesigned to take advantage of the full capabilities of modern hardware and begin to close the "deficit" between growing volumes of data and query complexity and the slower growth in single-core CPU performance. The project has had several major successes and key research results.

The first major research undertaking in the Quickstep project was the development of a unique flexible block-based storage engine. Table storage in Quickstep is divided into many self-contained, self-describing blocks that internally allow a variety of different physical storage organizations. An evaluation of the in-memory performance implications of several dimensions of physical storage organization (including row-store vs. column-store tuple layout, the use of indices, and the use of compression) was carried out in this framework. This sort of apples-to-apples performance evaluation of different storage design choices would not have been possible without the Quickstep storage manager, since even databases with a plug-in mechanism for different storage engines (e.g. MySQL) make assumptions about access patterns over data that bias them towards certain storage formats (e.g. row-at-a-time iteration via a cursor). Blocks in Quickstep, on the other hand, have a high-level logical relational API, and different blocks decide for themselves based on their own internal organization how to most efficiently evaluate operations like selection and projection.

The experimental study of in-memory storage organizations produced a number of interesting results, and the most important is this: there is no "one size fits all" storage organization that consistently outperforms the others. For instance, while column-stores are widely used as the primary or only layout for tuple data in many analytic DBMSes [65, 84, 92, 113] there are still a class of queries (particularly those that project multiple attributes and may involve indexing) where row stores have a performance edge. While some systems rely heavily on compression and deemphasize the use of indices or eliminate them all together [35, 84, 85, 92], both techniques can actually improve *or* diminish performance depending on query selectivity. The lack of a universally high-performing storage organization points to a need for flexibility in the physical organization of data, so that the right organization for a particular data set and work load can be chosen. Quick-

step's block-based storage architecture provides this flexibility. It must be acknowledged, however, that the chosing storage parameters in such an open environment is a significant challenge. The extensive results presented in Chapter 3, which include detailed explanatory information about cache behavior, are a useful basis for making such decisions.

Another key component of Quickstep is the communication framework, the *Transactional Message Bus* (TMB). Even the single-node version of Quickstep that is the primary focus of this work needs a mechanism for communication between the scheduler and worker threads that scales well in multi-socket NUMA systems. This component was also developed with an eye towards a future distributed version of Quickstep, so it was useful to clearly define the semantics of messaging and develop practical implementations of the TMB that would serve both the current need for in-process communication in Quickstep and provide the same features transparently over the network for future work (and indeed, for other applications).

The process of defining the TMB's semantics started by surveying the features of existing message-oriented middleware systems. Such systems provide desirable properties relating to reliability (e.g. at-least once delivery, persistent queues, high availability), consistency (e.g. virtual synchrony), and manageability (e.g. tied messages, priority and deadlines), but the development of the TMB was further motivated by the fact that no preexisting system delivered the complete set of these properties. The first key contribution of the TMB is a complete definition of messaging semantics incorporating all of these features.

The actual implemention of TMBs builds on Jim Gray's argument that queues are databases [44] and treats message bus management as a form of transaction processing. A modular TMB software architecture (with "pluggable" components for core bus management, persistence and recovery, and networking) was developed, with implementations based on existing database software (SQLite, LevelDB, Zookeeper, and VoltDB) as well as written-from scratch "native" implementations. The performance and scalability of these implementations was evaluated using a stress-test benchmark, which guided tuning and optimization of TMB components like in-memory concurrency control mechanisms, and revealed interesting properties of the various TMB implementations. One particularly interesting result is that, when threads running on different sockets in a NUMA server communicate with each other over a TMB, there is not a major performance drop from inter-socket communication so long as the threads themselves are affinitized to run on particular sockets. Another interesting result is that the distributed TMB implentations (both

the native TMB network server and TMB running on a VoltDB cluster) have a property of network-agnostic scalability, which is to say that the relationship between the number of clients (threads) using the TMB and the overall throughput is similar regardless of the number of physical machines those clients are spread over.

The last major contribution of this work is the development of Quickstep as a complete analytics engine building on the storage engine and TMB components. The query execution model in Quickstep is designed to exploit the parallelism of modern CPUs at multiple levels. At the lowest level, Quickstep incorporates bit-parallel indexing methods [70] to take advantage of the parallelism inherent in ALUs with wide word sizes. At a slightly higher level, Quickstep uses a system of code templates for expression evaluation that compile down to simple column-at-a-time loops that the compiler can transform to use SIMD instructions for instruction-level parallelism. Finally, to take advantage of ever-increasing numbers of CPU cores, Quickstep has a parallel and dynamic query execution framework that delivers both inter-operator and intra-operator parallelism.

The block-based storage architecture in Quickstep plays a crucial role in the parallel execution engine. Storage blocks are a natural unit of parallelism for multiple simultaneous worker threads. Simple operations like selection and projection can be performed completely independently for different blocks. More complex operations require some shared state between workers (e.g. building and probing hash-tables for joins and aggregation with grouping). Quickstep still achieves block-level parallelism for more complex operators by using latch-free in-memory data structures for shared state and allowing multiple workers to process different input blocks using these shared structures in parallel. Blocks are also the units of data flow between operators in query plans, so that intermediate results are materialized in blocks which can then be used as inputs for downstream operators, with different workers simultaneously executing block-level "work orders" from different operators in a query plan.

Quickstep has a dynamic scheduler, and does not attempt to statically set a degree of parallelism for queries up-front. Instead, the scheduler opportunistically schedules whatever block-level work orders are available at a given moment using any available stateless worker threads. As intermediate blocks are materialized by executing work orders, new work orders are generated for downstream operators in the query plan, and these too are opportunistically dispatched to any available workers. This dynamic multithreaded execution model achieves excellent scaling with addition worker threads, as demonstrated by benchmarking experiments in Chapter 5, and can even achieve instant elasticity for a

running query as new resources become available.

Despite these successes, it must be acknowledged that challenges and open questions remain for Quickstep. For instance, experiments in Section 5.2 showed that Quickstep's performance for large joins can be underwhelming compared with another leading main-memory database, MonetDB. The hash-based algorithms used for joins and aggregation in Quickstep need to be tuned to use CPU caches efficiently (much as the study of storage organizations in Chapter 3 provided insights for tuning selection algorithms). The current Quickstep prototype also runs on only one node, and a major long-term goal for the project is to develop a distributed version that extends the scalability and elasticity of the Quickstep execution engine across multiple nodes in a cluster or cloud setting, allowing Quickstep to handle workloads and data sets that are too large for any one server.

Fortunately, the Quickstep project continues apace with contributions from both student researchers and commercial developers. The issues and goals mentioned above, along with many others, continue to be worked on by the thesis author and many colleagues. Quickstep's future as both a research platform and a practical database product is bright.

BIBLIOGRAPHY

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682, 2006.

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

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

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 169–180, 2001.

[5] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB*, pages 1064–1075, 2012.

[6] Amazon Web Services, Inc. Amazon Simple Queue Service. `https://aws.amazon.com/sqs/`, 2014.

[7] Apache Software Foundation. ActiveMQ. `https://activemq.apache.org/`, 2014.

[8] Apache Software Foundation. Apache Kafka: A high-throughput distributed messaging system. `https://kafka.apache.org/`, 2014.

[9] Apache Software Foundation. Apache Zookeeper. `https://zookeeper.apache.org/`, 2014.

[10] Apache Software Foundation. The Hadoop Distributed File System. `https://hadoop.apache.org`, 2015.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394, 2015.

[12] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[13] C. Balkesen, J. Teubner, G. Alonso, and M. T. Oszu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. *ICDE*, 2013.

[14] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.*, 27(7):1754–1766, 2015.

[15] G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18, London, UK, UK, 1999. Springer-Verlag.

[16] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business analytics in (a) blink. *ICDE*, pages 9–14, 2012.

[17] R. Barber, G. M. Lohman, I. Pandis, V. Raman, R. Sidle, G. K. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *PVLDB*, 8(4):353–364, 2014.

[18] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 112–122, New York, NY, USA, 1990. ACM.

[19] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[20] K. P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, 1985. ACM.

[21] K. P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, Dec. 1993.

[22] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 3–, New York, NY, USA, 1983. ACM.

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

[24] S. Blanas and J. M. Patel. Memory footprint matters: efficient equi-join algorithms for main memory data processing. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 19:1–19:16, 2013.

[25] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, Dec. 2008.

[26] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[27] L. Bouganim, D. Florescu, and P. Valduriez. Multi-join query execution with skew in NUMA multiprocessors. In *13ème Journées Bases de Données Avancées, 9-12 septembre 1997, Grenoble (Informal Proceedings).*, 1997.

[28] D. Campbell. Service oriented database architecture: App server-lite? In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 857–862, New York, NY, USA, 2005. ACM.

[29] C. Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD*, pages 355–366, 1998.

[30] E. Curry. Message-oriented middleware. *Middleware for communications*, pages 1–28, 2004.

[31] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[32] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. *The Benchmark Handbook for Database and Transaction Systems. Morgan Kaufmann*, 1993.

[33] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. pages 1243–1254, 2013.

[34] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[35] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *SIGMOD*, pages 45–51, 2011.

[36] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[37] Z. Feng and E. Lo. Accelerating aggregation using intra-cycle parallelism. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 291–302, 2015.

[38] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 31–46, 2015.

[39] Google Inc. Google Compute Engine. `https://cloud.google.com/products/compute-engine/`, 2014.

[40] Google Inc. LevelDB. `https://code.google.com/p/leveldb/`, 2014.

[41] Google Inc. GRPC. `http://www.grpc.io/`, 2015.

[42] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 102–111, New York, NY, USA, 1990. ACM.

[43] G. Graefe. Sorting and indexing with partitioned b-trees. *CIDR*, 2003.

[44] J. Gray. Queues are databases. In *Proceedings of the 7th High Performance Transaction Processing Workshop*, Asilomar, CA, USA, 1995.

[45] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, Sept. 1996.

[46] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. *VLDB*, pages 105–116, 2010.

[47] R. A. Hankins and J. M. Patel. Data morphing: an adaptive, cache-conscious storage technique. *VLDB*, pages 417–428, 2003.

[48] Hapner, Mark and Burridge, Rich and Sharma, Rahul and Fialli, Joseph and Stout, Kate and Deakin, Nigel. Java Message Service. `https://jcp.org/aboutJava/communityprocess/final/jsr343/index.html`, 2013.

[49] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. *VLDB*, pages 487–498, 2006.

[50] M. Henning. The rise and fall of corba. *Queue*, 4(5):28–34, June 2006.

[51] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *VLDB*, pages 502–513, 2008.

[52] IBM Corporation. WebSphere MQ. `http://www-03.ibm.com/software/products/en/websphere-mq`, 2014.

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

[54] iMatix Corporation. ZeroMQ. `http://zeromq.org/`, 2014.

[55] Intel Corporation. Intel xeon processor e7-4800/8800 v3 product families: Datasheet volumn1: EMTS, May 2015.

[56] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. VERTEXICA: your relational friend for graph analytics! *PVLDB*, 7(13):1669–1672, 2014.

[57] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[58] T. Johnson. Performance measurements of compressed bitmap indices. In *VLDB*, pages 278–289, 1999.

[59] 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, Aug. 2008.

[60] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[61] T. Kiefer, T. Kissinger, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS live: a numa-aware in-memory storage engine for tera-scale multiprocessor systems. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 689–692, 2014.

[62] T. Kiefer, B. Schlegel, and W. Lehner. Experimental evaluation of NUMA effects on database management systems. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pages 185–204, 2013.

[63] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *VLDB*, pages 1378–1389, 2009.

[64] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[65] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *VLDB*, pages 1790–1801, 2012.

[66] L. Lamport. Multiple byte processing with full-word instructions. *Commun. ACM*, 18(8):471–475, 1975.

[67] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[68] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754, 2014.

[69] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[70] Y. Li and J. M. Patel. Bitweaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 289–300, New York, NY, USA, 2013. ACM.

[71] Y. Li and J. M. Patel. WideTable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[72] P. E. McKenney and J. Walpole. Introducing technology into the linux kernel: A case study. *SIGOPS Oper. Syst. Rev.*, 42(5):4–17, July 2008.

[73] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB*, pages 53–72, 1997.

[74] C. Molina-Jimenez, S. Shrivastava, and N. Cook. Implementing business conversations with consistency guarantees using message-oriented middleware. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '07, pages 51–, Washington, DC, USA, 2007. IEEE Computer Society.

[75] B. Murthy, M. Goel, A. Lee, D. Granholm, and S. Cheung. Oracle exalytics in-memory machine: A brief introduction. `http://www.oracle.com/us/solutions/ent-performance-bi/business-intelligence/exalytics-bi-machine/overview/exalytics-introduction-1372418.pdf`, October 2011.

[76] D. R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993, 1997.

[77] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE*, pages 183–194, 2011.

[78] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.pdf`, Jan 2007.

[79] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49, 1997.

[80] Oracle Corporation. MySQL. `https://www.mysql.com/`, 2015.

[81] Pivotal Software, Inc. RabbitMQ. `http://www.rabbitmq.com/`, 2014.

[82] Progress Software Corporation. Sonic MQ. `http://www.progress.com/products/openedge/solutions/application-integration/aurea-sonic-mq`, 2014.

[83] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.

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

[85] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. *ICDE*, pages 60–69, 2008.

[86] S. Ramani, K. S. Trivedi, and B. Dasarathy. Reliable messaging using the corba notification service. *Distributed Objects and Applications, International Symposium on*, 0:0229, 2001.

[87] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. *SIGMOD*, pages 475–486, 2000.

[88] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. *SIGMOD*, pages 558–569, 2002.

[89] D. Rinfret, P. E. O'Neil, and E. J. O'Neil. Bit-sliced index arithmetic. In *SIGMOD*, pages 47–57, 2001.

[90] K. A. Ross and K. A. Zaman. Serving datacube tuples from main memory. In *Scientific and Statistical Database Management*, pages 182–195. IEEE, 2000.

[91] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. *SIGMOD*, pages 23–34, 1979.

[92] 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*, pages 731–742, 2012.

[93] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: a modular query optimizer architecture for big data. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 337–348, 2014.

[94] Spread Concepts LLC. Spread Toolkit. `http://www.spread.org/`, 2014.

[95] SQLite Consortium. SQLite. `https://sqlite.org/`, 2014.

[96] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. *VLDB*, pages 553–564, 2005.

[97] S. Tai, T. Mikalsen, I. Rouvellou, and S. M. S. Jr. Conditional messaging: Extending reliable messaging with application conditions. *2002 IEEE 22nd International Conference on Distributed Computing Systems*, 0:123, 2002.

[98] S. Tai and I. Rouvellou. Strategies for integrating messaging and distributed object transactions. In *IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware '00, pages 308–330, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[99] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[100] Typesafe Inc. Akka. `http://akka.io/`, 2014.

[101] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 39(4):76–83, Apr. 1996.

[102] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, Feb 1997.

[103] VoltDB Inc. VoltDB. `http://voltdb.com/`, 2014.

[104] H. S. Warren. Functions realizable with word-parallel logical and two's-complement addition instructions. *Commun. ACM*, 20(6):439–441, 1977.

[105] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing database column scans with complex predicates. In *ADMS*, pages 1–12, 2013.

[106] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[107] K. M. Wilson and B. B. Aglietti. Dynamic page placement to improve locality in cc-numa multiprocessors for tpc-c. In *ACM/IEEE Conference on Supercomputing*, pages 33–33, 2001.

[108] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.

[109] M.-C. Wu. Query optimization for selections using bitmaps. In *SIGMOD*, pages 227–238, 1999.

[110] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[111] Q. Zeng, J. M. Patel, and D. Page. Quickfoil: Scalable inductive logic programming. *PVLDB*, 8(3):197–208, 2014.

[112] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[113] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.