Exploration of a Vision for Actor Database Systems

Vivek Shah
DIKU, Department of Computer Science
University of Copenhagen, Denmark

July 31, 2017

PhD Dissertation
This dissertation has been submitted to the PhD School of the Faculty of Science, University of Copenhagen, Denmark

Abstract

We have witnessed an unprecedented growth in the variety and volume of interactive data intensive web services over the past decade owing to improving Internet connectivity and increasing penetration of smartphones and sensors. Similar growth has also been noticed in the evolution of computing hardware. Machines with multiple processor sockets and cores and large main-memory are now commonplace. Cloud computing infrastructures have also gained widespread adoption. As a result, these are exciting times to design high-performance scalable software systems to enable productive development and deployment of interactive data intensive services.

Existing popular approaches to build these services can be classified under two extreme ends. At one end, a modern high-performance in-memory database system is employed as a full-fledged programming environment to both store data and execute application logic. At another end, an actor runtime is utilized to store application logic and data in memory while utilizing a key-value storage layer only for persistence. Curiously, we notice that both these disparate approaches have complementary strengths and weaknesses. While database systems provide transactional guarantees along with a high-level data model supporting declarative query capabilities as robust and elegant state management features, they lack programming model primitives for modularity, scalability and to exercise performance control. While actor runtimes provide the actor primitive for modularity and scalability as well as an asynchronous messaging construct to exercise performance control, they lack robust and elegant state management features.

In this dissertation, we propose a new vision of an *actor database system* in order to integrate actor programming model features in database systems. The purpose of this dissertation is to explore this vision of an actor database system for its viability and benefits. Towards this goal, we make three main contributions. First, we argue for a new vision of actor database systems in light of current application and hardware trends. We concretize the notion of an actor database system by defining its feature sets followed by a detailed case study of a new benchmark, SmartMart, which is representative of emerging interactive data intensive services.

Second, we introduce a new programming model for actor database systems called *reactors* that enriches relational databases with an actor construct and support for asynchronous function invocations under ACID guarantees. We build an in-memory multi-core OLTP database engine, named ReactDB, and demonstrate

the scalability of the system and its ability to translate the performance benefits of the programming model by evaluating it under classic OLTP benchmarks.

Third, we investigate the potential of leveraging task-level parallelism available in complex application logic using the reactor programming model in ReactDB on existing multi-core hardware. Using the SmartMart benchmark, we vary and observe the effects of a number of classical parallelism factors in order to demonstrate the effectiveness of the programming model and the low-overhead implementation of transactional parallelism in the system.

Resumé

Det forgangne årti har medbragt en hidtil uset vækst i forskelligartede og og mangfoldige interaktive data-intensive web-tjenester, i særdeleshed grundet forbedret adgang til Internettet, samt spredningen af smartphones og sensorer. En lignende vækst er fundet sted indenfor udviklingen af datamaskiner. Det er blevet almindeligt med maskiner udstyret med adskillige processorer og kerner, samt store mængder af hukommelse. Brugen af cloud computing er også blevet hverdag. Følgeligt er det interessant at studere hvorledes man bør designe højtydende og skalérbare systemer med henblik på effektiv udvikling og anvendelse af interaktive data-intensive tjenester.

Gængse fremgangsmåder til konstruktion af sådanne tjenester kan klassificeres i to yderkategorier. I den ene kategori finder vi moderne højtydende in-memory databaser, der kan benyttes som fuldgyldige programmeringsmiljøer der både lagrer data og afvikler programlogik. I den anden kategori finder vi anvendelsen af en actor runtime til at lagre programlogik og data i hukommelsen, hvor et traditionelt lagersystem kun bruges til at sikre persistens. Disse uensartede fremgangsmåder viser sig at have styrker og svagheder der komplementerer hinanden. Databasesystemer understøtter transaktionelle garantier og abstrakte datamodeller med deklarative forespørgsler, samt en robust og elegant teknik til tilstandshåndtering, men programmeringsmodellen savner primitiver til understøttelse af modularitet, skalérbarhed, samt detaljeret operationel kontrol. Medens actor runtimes netop har aktørmodellen som fundament for modularitet og skalérbarhed, samt asynkrone beskeder til operationel kontrol, så savner de en robust og elegant teknik til tilstandshåndtering.

I denne afhandling præsenterer vi en vision for et *actor database system*, der netop integrerer funktionalitet fra aktørmodellen i databasesystemer. Formålet med denne afhandling er at udforske anvendeligheden og fordelene ved et sådan actor database system. I dette regi gør vi tre konkrete bidrag. For det første argumenterer vi for actor database systems i lyset af de nuværende trends indenfor applikations- og maskinudvikling. Vi konkretiserer idéen med et actor database system ved at definere dets funktionalitet, fulgt af et detaljeret studie af et nyt benchmarkprogram, SmartMart, som er repræsentativt for kommende interaktive data-intensive tjenester.

For det andet beskriver vi en ny programmeringsmodel for actor database systems, kaldet *reactors*, der beriger relationelle databaser med en aktørkonstruktion, samt understøttelse for asynkrone funktionskald med ACID-garantier. Vi konstruerer en in-memory flerkernet OLTP-databasemotor, kaldet REACTDB, og viser skalérbarheden af systemet, og dets evne til at realisere ydelsesfordelene ved programmeringsmodellen, ved en evaluering med klassiske OLTP-benchmarkprogrammer.

For det tredje undersøger vi potentialet for, på eksisterende flerkerne-maskiner, at udnytte den job-parallelisme der er til rådighed i kompleks programlogik udtrykt via reactor-modellen in REACTDB. Med SmartMart som arbejdsbyrde ændrer og observerer vi effekten af en række klassiske faktorer indenfor parallelisme, med henblik på at vise programmeringsmodellens effektivitet, og den effektive implementering af transaktionel parallelisme i systemet.

To my family

Contents

List of Figures

List of Algorithms

Acknowledgments

I would first and foremost like to acknowledge my primary advisor Marcos Antonio Vaz Salles and co-advisor Fritz Henglein. Marcos has been my advisor, mentor and friend for the past four years. I am thankful to him for introducing me to the field of database systems and inspiring me to pursue my graduate studies in it. He has taught me the value of perseverance, dedication and optimism in the pursuit of "that elusive nugget" of simplicity and elegance within a complex mess. I am grateful to him for inspiring me to dare and for pushing me to settle for nothing less than excellence in research. He has provided me with unending support and deep technical insight whenever I needed it, while allowing me the utmost freedom to delineate my own research agenda. I am also grateful to him for his meticulous commentary on this document. In hindsight, I could not have asked for a better advisor. I am especially thankful to Fritz for providing me the support and freedom during my graduate studies and being inspirational as always.

I am also grateful to Phil Bernstein for inviting me to Microsoft Research for an internship in the Orleans project. My discussions with Phil helped me gain understanding of actor systems and their challenges and greatly motivated my work. It has been both a humbling and inspiring experience to witness his passion for knowledge. I will always cherish his memoirs of database history that he shared over the course of our lunch conversations.

I have been fortunate to spend my time at DIKU with some wonderful researchers. I would like to thank all the members of the DIKU APL section for all the active and fun discussions and for providing me with a pleasant and stimulating work environment. To my friends at DIKU, especially Ulrik, Bjorn, Robin, Danil, Kostas, Noy and Trine, a special thank you for all the memories and the fun times together. I would also like to thank the administrative staff, especially Jette, for all the help and support. I am thankful to my friend Tim for proofreading parts of this dissertation. I am grateful to Troels for his critical help in translating the enclosed Danish abstract.

I am eternally grateful to my parents and to my little sister, Princy, for all the love and support they have provided me throughout my life and for always believing in me despite the odds.

Finally, I would like to thank my wife Khushboo for everything and beyond, without whom none of this would have been possible.

Chapter 1

Introduction

"Imagination is more important than knowledge."

— Albert Einstein

1.1 Motivation

The past decade has witnessed an explosion in the growth of interactive data-intensive web services [47, 65]. Buoyed by ever growing Internet connectivity, proliferation of smartphones and sensors, and an ever-growing human desire to digitize our interaction with the world around us, the variety and volume of interactive data-intensive web services continues to grow. This growth has been fueled by the increase in availability of tools to build these services and the improving accessibility and usability of cloud computing infrastructure for their deployment. Interactive data-intensive web services span diverse domains ranging across classic banking, reservation, telecommunication and order entry systems, to newer and emerging domains of multiplayer online games, social networks, financial fraud detection, portfolio risk assessment, online trading, information visualizations, chat and call services, crowd collaboration and synchronization platforms, mobile and social payment platforms, real-time bidding applications and even IoT streaming from space balloons [7,31,57,63,93,96,97,105,120,132]. This rise in application diversity is only expected to increase given the predicted penetration of smartphones, Internet connectivity and IoT devices. The increase in the variety of interactive data-intensive web services has also gone hand in hand with the rise of their volume i.e., the number of users of these data-intensive web services, which has raised the scalability bar of these services, especially ones that become popular.

In order to build, maintain and deploy these services, various approaches exist in the design space. Before categorizing the approaches in the design space of these services, we discuss the service domain trends in Section 1.1.1 and the hardware trends in Section 1.1.2 that are being witnessed over the past decade. We discuss these trends to highlight their impact on the software systems that are being used to build and deploy these services.

1.1.1 Interactive Data Intensive Service Domain Trends

One of the distinctive traits of modern services is their interactive nature that enforces hard real-time requirements to elicit a response. Interactivity and hard real-time requirements stem either from these services being user facing or from a need to respond in real time to devices (e.g. to sensors sending readings). One of the central goals of these web services today is to first minimize the time to market in order to capture market share of users. This is followed by the requirement to minimize time to add new features to the service and to fix bugs. This is necessary to retain current users and to attract new users in the face of competitors in an increasingly hostile online marketplace. As a result, it is imperative that as the application complexity (size of code base, data footprint) increases, it remains easy to maintain. This mandates that the software systems used to build and deploy these services must maximize programming productivity.

As the service becomes popular, it is critical that they scale to support the number of growing users without affecting the performance perceived by individual users. This directly puts the onus on the application developers to either ensure that the design of the service is scalable to start with, or to understand the scalability issues in the service and redesign it. Even though a service might be scalable, in the presence of competition the application developers need to reason about and improve the service performance if possible. Thus, this requires that the software systems employed to build these services must be scalable, should provide their users with features to understand service scalability and performance, and must allow the flexible control of these features.

The workload of these services has traditionally resembled classic on-line transaction processing workloads [28, 61, 129], which have minimal application logic, span a few records with key-based lookups, are write-heavy and require consistency. However of late, especially with emerging new domains the amount and nature of application logic interacting with the data has changed. This change in application logic has come or is anticipated to come in various forms. Heterogeneous workloads consisting of read-mostly transactions [128] are being increasingly observed that mix scans and aggregation operations over small ranges in the traditional OLTP workload of point look-ups. Increasingly, more conditional statements influencing different data and query footprints are being observed in these workloads. As newer services emerge by modeling non-traditional application domains, complex application logic can range across diverse domains such as financial computations [33], real-time simulations [127, 130, 131], and online machine learning methods [90], to name a few.

1.1.2 Hardware Trends

The past decade has also witnessed enormous growth on the hardware front. With gains diminishing from single-threaded performance of processors, chip designers are increasing the number of cores on a processor socket and the number of processor sockets in order to provide more computation power. In line with Moore's law, the processors with increasing core counts and sockets are becoming cheaper as well. Similar gains have been observed in main-memory capacity which has accompanied a decrease in their price. The authors of [121] argued ten years back that most OLTP applications would fit the main memory of either a single node machine or a cluster of machines. Their argument is as follows

> "The overwhelming majority of OLTP databases are less than 1 Tbyte in size and growing in size quite slowly. For example, it is a telling statement that TPC-C requires about 100 Mbytes per physical distribution center (warehouse). A very large retail enterprise might have 1000 warehouses, requiring around 100 Gbytes of storage, which fits our envelope for main memory deployment."

Now, a machine with 256 GB RAM and two sockets each with eight physical cores costs around 8000 USD which would suffice for the deployment argued above. The same deployment can be achieved at an even lower cost with a cluster of cloud computing nodes. Similar price/performance gains have been observed in disk technology (solid state drives) and in networking infrastructure (RDMA) as well. Cloud computing as an infrastructure is also seeing widespread adoption for the deployment of these data intensive services. All these hardware trends require that the software systems used to deploy these services need to be extremely resource efficient in order to translate the hardware benefits to perceivable application performance [10].

1.1.3 Interactive Data Intensive Service Design Approaches

Existing popular approaches of building interactive data intensive services employ a three-tier client-server architecture varying the utilization of the data and the application tiers in different ways. All these approaches can be broadly classified under two extreme ends outlined below:

1. Utilize an in-memory OLTP database system as a full-fledged programming environment where the entire application logic and data resides while the application tier is completely stateless.

2. Utilize an application programming framework (actor frameworks) in the application tier where the entire application logic and data resides while a key-value storage layer is employed at the data tier only for persistence.

In Sections 1.2 and 1.3, we highlight the benefits and limitations of both these design approaches in light of the previously identified software and hardware trends. In fact, we observe that the benefits and limitations of these two extreme approaches are complementary to each other. As a result, we highlight the importance of the opportunity to explore the integration of the feature sets of these two disparate system domains. This exploration forms the crux of this dissertation, which explores the design space of the integration of actor programming models in modern database systems. The dissertation investigates the viability and potential of such a system by building an in-memory database system for multi-core machines exposing an actor programming model and then by evaluating the system and the programming model for benchmarks representative of interactive data intensive services.

1.2 Modern In-Memory OLTP Database Systems

In light of the hardware trends outlined in Section 1.1.2, the database community made concerted efforts to identify scalability and performance issues in traditional disk-based OLTP database designs for multi-core and multi-machine deployments [68,75,76,121]. Given these findings, extensive work has been done by the community to redesign the database system internals for main-memory, multi-core and multi-machine deployments [50,79,80,92,98,125]. The theme of this work has been on increasing the scalability of modern in-memory database systems to high transaction volumes by either removing the bottlenecks in the traditional systems or by rebuilding them from scratch.

While the internals of these systems were redesigned, the classical abstraction provided by databases remained unchanged. The classic guarantees of ACID transactions for execution of application programs makes it very easy to prototype applications using these systems. The high-level data model and declarative query capabilities simplify how the application logic accesses and manipulates data. Atomicity guarantees free the application developer from reasoning about partial executions of programs either due to program errors or failure of the database system. The guarantees of serial execution of programs (serializable isolation level) completely insulates application developers from the effects of concurrency. The durability guarantee ensures that the effects of successfully completed programs are recoverable. All these features in conjunction greatly simplify nascent design of interactive data-intensive services thus making them an ideal environment to quickly prototype applications.

However, as the application complexity grows due to increase in size of code base (stored procedures), the number of relations, constraints and triggers, it becomes very hard to debug and isolate bugs and failures because a tuple in a relation can be modified by any stored procedure (lack of data encapsulation makes it hard to trace source of bugs). The lack of modularity features in these systems makes it hard to tame the complexity of the service over time. Database systems have traditionally only provided a data model while the programming model has been left implicit. This implicit programming model provides a monolithic view of the database, i.e., a machine with a single unit that stores data and a single computational unit that executes queries to access and manipulate data sequentially. Since the programming model lacks the notion of a scalable computational entity, it is hard if not impossible to reason about the scalability of programs executing on it. As a result, designers of interactive data intensive services often find it difficult to isolate whether scalability issues arise from the application logic in the service or from scalability issues of the database system.

A stored procedure is used to implement the application logic that executes sequentially one statement at a time in these systems. Stored procedure statements can consist of a sequence of queries interspersed with conditional statements, loops and function calls. The query optimizer in a database system can only infer data-flow parallelism in a single query and hence can only parallelize individual queries within a stored procedure. In addition, there is no guarantee that the query optimizer will infer or choose to exploit data-flow parallelism in a query. The lack of a scalable computational primitive in the programming model restricts the developer from specifying parallelism across multiple statements in stored procedures. As the complexity of application logic in interactive data intensive application grows, application developers will increasingly feel the lack of necessary primitives to control and improve the performance of their programs especially in the parallel hardware available today.

Modern in-memory OLTP database systems are designed either using a shared-everything [50, 76, 125] or a shared-nothing architecture [80, 121]. Data and code partitioning is a necessity in a shared-nothing database due to their architecture to distribute the code and data across the partitions. Some shared-everything databases that employ internal partitioning of data structures for high performance in multi-core machines [98] also need to perform data and code partitioning. Since the programming model does not have any notion of partitioning, shared-nothing databases depend on inference mechanisms to distribute code and data across partitions [41, 101]. The performance of these systems is predicated on the inference of a good partitioning layout. Since these inference mechanisms depend on the workload and the database implementation, their fragility leads to unpredictable performance behaviors that cannot be cleanly understood and influenced using the programming model [72]. Even the high performance of shared-everything database systems that do not necessitate data partitioning [125] in a multi-core setting is reliant on the affinity of data to the processing cores because of cache coherence and NUMA effects, and any performance variability cannot be explained and controlled using the programming model guarantees.

Summary. The robust state management features of the modern in-memory database systems aid in the design of interactive data intensive services by freeing the application developer from reasoning about correct manipulation of state under concurrency and failures, thus allowing her to focus on the specification of the application logic in a high-level data model with declarative query capability. However, the lack of modular programming abstractions complicates debugging and isolation of faults especially as the size of the service (data and code base) grows with time. The lack of a computational primitive in the programming model hinders reasoning about scalability and exercising flexible control over the application logic to improve performance.

1.3 Actor Systems

The use of application frameworks for designing interactive data intensive applications has been gaining a lot of attention [18, 118, 119]. Actor programming models [4, 23] are seeing increased adoption for deploying these services in cloud computing infrastructures [6, 56, 97, 110]. The distributed actor programming model provides the scalable computational construct of an actor to allow a service to be constructed using this building block. The distributed programming model enables reasoning about the scalability of the service in terms of the actor building block. By controlling the granularity of the actors, the application developer can flexibly control the scalability of her service. The distributed programming model obviates the need for partitioning in order to deploy the service in cloud computing infrastructures.

By encapsulating state within the actor and enforcing restrictions on the access of state across actors, they provide the benefits of a modular abstraction. In addition, programming language support for modularity in these systems greatly aids in isolating bugs and failure, and testing the service as it grows in complexity and size. The popularity of microservices [69] highlights the importance of a loosely coupled design for modularity benefits. Actor programming models enable the application developer to flexibly define the granularity of their modules and achieve the most beneficial module architecture. Actor programming models also provide a computational construct in addition to a modular one. This computational construct along with asynchronous message passing semantics of actors makes it possible for application developers to specify and leverage parallelism within the application logic. As a result, the application developers can reason about and flexibly control the performance of the service. This is especially important given the service trends of increasing complexity in application logic.

Despite their appeal, actor runtimes such as Erlang and Akka [6, 14] expose problems of actor lifetime management and sharing their references in order to communicate to the application developer. This becomes a hard problem to solve especially in a cloud computing infrastructure. Orleans [23] obviates these problems by removing these responsibilities from the service and delegating it to the run-time. Nevertheless, other problems of state management remain. Although actors provide the notion of single-threaded execution within an actor, management of actor state for durability is exposed to the application developers. These systems typically employ a key-value storage layer [38] for durability of actor state that needs to be explicitly managed by the application developer. Additionally, consistency of state across actors under concurrent execution of programs and under failures (errors in application logic, failure of the system) needs to be managed by application developers. This leads to complications in application logic for state management and impacts developer productivity. The data model in these systems is tied to either the actor programming language run-time or the programming language used to build the actor framework. As a result, there is no physical data independence and consequently portability of the service suffers across actor systems. In addition, the lack of declarative query capabilities also affects developer productivity. The lack of these features has also motivated a call to integrate database functionality into actor run-times and work is underway in this direction [26, 54, 124].

Summary. Distributed actor programming models provide the desirable primitive of a modular, scalable computational construct that can be used to reason about modularity and scalability of interactive data intensive services. This puts the application developer firmly in control to tame complexity and scalability issues as the service grows in size. Asynchronous messaging between actors provides a construct to reason about the performance of application logic and to control it. The lack of robust state management under failures and across actors either limits the use of these actor systems to services that can tolerate these issues or raises the need to handle the issues in the service. This leads to increased complexity of the service and lowers developer productivity. The lack of a high-level data model and declarative query capabilities affects portability of these services and curtails any benefits that could be obtained from physical data independence and optimization of query plans for declarative queries.

1.4 Summary of Goals and Contributions

In light of the discussion of the previous sections, the overall goal of this dissertation is to concretize the idea of an actor database system and investigate its viability and benefits. Consequently, our work has been organized under three functional blocks namely (1) to outline the vision of an actor database system in light of current application and hardware trends (2) to investigate the challenges that arise in the design and implementation of an actor database system by building an in-memory multi-core OLTP database engine with an actor programming model (3) to evaluate whether the performance guarantees of the programming model can be leveraged for performance gains in the system, especially for benchmarks with available task-level parallelism on multi-core machines. We elaborate the contributions of each of these functional blocks below:

1.4.1 Actor Database Systems Vision

In the first functional block, we motivate the need for actor database systems by analyzing how existing approaches of building interactive data intensive services do not address the observed application and hardware trends. During our analysis of the shortcomings of existing approaches, we also emphasize how actor database systems would address these issues. In light of the aforementioned analysis, we concretely define an actor database system by postulating their mandatory feature sets, which are organized under four broad tenets. These tenets identify the characteristics of an actor database system which are given a precise interpretation by the feature sets. To highlight the necessity and applicability of the feature sets, we perform a detailed case study for a benchmark (SmartMart) that we designed to model an interactive data-intensive smart supermarket application. In the case study we use detailed application pseudocode to demonstrate how the promulgated features of an actor database system can be utilized by an interactive data intensive application. In addition to demonstrating the benefits of the actor database system features by pseudocode examples, we also illustrate the potential of the programming model in improving performance by evaluating the benchmark in a prototype system discussed in the next section.

1.4.2 Reactor Programming Model and ReactDB

In the second functional block, we investigate the challenges that arise in order to integrate actor programming models inside database system kernels on machines with large main memory and multi-core processors. In order to validate the viability of the system and the programming model, we enrich the relational database programming model with (1) the computational construct of an actor and (2) asynchronous function invocation semantics. This new programming model named *reactors* guarantees atomic and serializable execution of programs. In order to support the reactor programming model, we build a multi-core in-memory OLTP database engine named ReactDB that comprises of in-memory storage, concurrency control and transaction coordination layers. In order to focus on the challenges of integrating reactor programming model in the kernel of a high-performance database engine and to understand whether the performance promises of the programming model can actually be perceived in practice, we refrain from building a full-feature actor database system, thus delegating support for declarative querying, durability, monitoring and administration to future work. We discuss in detail the design and implementation of ReactDB highlighting how the system virtualizes the database architecture that can be configured when the system is deployed. In our results with classic OLTP benchmarks we show that the reactor programming model can be leveraged for observable performance gains at the microsecond scale. In addition, we also demonstrate the scalability of the system under different configurations of database architecture.

1.4.3 Transactional Parallelism in Reactors and ReactDB

In the third and final functional block, we build upon our previous evaluation work of classic OLTP benchmarks discussed in the previous section. We use the SmarMart benchmark that was designed in the first functional block (Section 1.4.1) with controllable task-level parallelism in mind to systematically study how classic parallelism factors of overhead, parallelizable work and its dependencies, interference and skew affect the performance of ReactDB. We also explain how the observed effects can be cleanly understood using the reactor programming model. We also conduct a detailed discussion on the internal implementation details of ReactDB to highlight their efficient design that plays a key role in ensuring parallelism benefits leveraged by application programs can be perceived in practice.

1.4.4 Publications

Previous versions of some of the chapters of this dissertation have been published or are in submission in international conferences. We list these chapters along with their respective publications below:

- **Chapter 2 - Actor Database Systems : A Manifesto [112]** Vivek Shah and Marcos Antonio Vaz Salles. Manuscript in Submission, preprint available as CoRR abs/17707.06507, 2017.

- **Chapter 3** consists of two publications namely:

    1. **Reactors: A Case for Predictable, Virtualized Actor Database Systems [113]** Vivek Shah and Marcos Antonio Vaz Salles. Manuscript In Submission, preprint available as CoRR abs/1701.05397, 2017.
    2. **Transactional Partitioning: A New Abstraction for Main-Memory Databases [111]** Vivek Shah. In VLDB PhD Workshop, 2014.

- **Chapter 4 - An Evaluation of Intra-Transaction Parallelism in Actor Database Systems.** Vivek Shah and Marcos Antonio Vaz Salles. Manuscript In Preparation, 2017.

1.4.5 Structure of Dissertation

The dissertation has been organized following the order of the functional blocks outlined in the contributions. In Chapter 2, we propose the vision of actor database systems arguing for its need and enunciating the feature sets that would identify an actor database system. We also perform a case study to exemplify the feature sets and performance potential of actor database systems in the context of an interactive data intensive application. In Chapter 3, we present a concrete programming model designed for actor database systems called reactors. We also discuss the design and implementation of an in-memory multi-core OLTP database system that exposes the reactor programming model named REACTDB. In Chapter 4, we discuss in detail the algorithms and mechanisms used by REACTDB to ensure that the reactor programming model can be leveraged for performance benefits with minimal overheads. We also perform a detailed evaluation complementing our previous work to highlight how classical parallelism factors manifest in REACTDB for an interactive, data-intensive application benchmark. Finally, in Chapter 5, we present our concluding remarks and outline ongoing and future directions of research.

Chapter 2

Actor Database Systems: A Manifesto

> *"Opposites are not contradictory but complementary."*
>
> — N. Bohr

Interactive data-intensive applications are becoming ever more pervasive in domains such as finance, web applications, mobile computing, and Internet of Things. Typically, these applications are architected to utilize a data tier for persistence. At one extreme, the data tier is a simple key-value storage service, and the application code is concentrated in the middle tier. While this design provides for programmability at the middle tier, it forces applications to forego classic data management functionality, such as declarative querying and transactions. At the other extreme, the application code can be colocated in the data tier itself using stored procedures in a database system. While providing rich data management functionality natively, the resulting lack of modularity and state encapsulation creates software engineering challenges, such as difficulty in isolation of bugs and failures or complexity in managing source code dependencies. In addition, this monolithic architectural style makes it harder to scale the application with growing request volumes and data sizes. In this chapter, we advocate a new database system paradigm bringing to developers the benefits of these two extremes, while avoiding their pitfalls. To provide modularity and reasoning on scalability, we argue that data tiers should leverage the actor abstraction; at the same time, these actor-based data tiers should offer database system features to reduce bugs and programming effort involved in state manipulation. Towards this aim, we present a vision for *actor database systems*. We analyze current trends justifying the emergence of this abstraction and discuss a set of features for these new systems. To illustrate the usefulness of the proposed feature set, we present a detailed case study inspired by a smart supermarket application with self-checkout.

2.1 Introduction

Online services are becoming increasingly ubiquitous requiring management of substantial data along with low-latency interactions. These interactive data-intensive applications include online games, social networks, financial systems, operational analytics data management, web applications and upcoming Internet-of-Things (IoT) and mobile computing platforms [31,33,63,120,132]. The standard methodology to architect these applications is to segregate the server-side application logic across an application (or middle) tier and a data tier. Opinions seem to be divided on how to architect the application code across the application and the data tiers. Existing approaches can be classified across two extreme ends: (1) Architect the data tier as a dumb storage abstraction with the entire logic in the application tier; (2) Architect the application tier to be completely stateless using the database as a full-fledged programming enviroment with the entire data manipulation logic in the data tier.

In approach (1), database functionality such as transactions and declarative querying are either sacrificed or underutilized. This approach leads to increased demand on the middle tier for well-defined state management functionality under concurrency and failures. This demand can be met either by: (a) building the necessary features in the application tier, however leading to reduced productivity [19, 115]; or (b) utilizing application frameworks with only limited transactional features in the middle tier, however leading to misconceptions and, consequently, incorrect applications [18].

In approach (2), the database is conceptualized as a monolithic entity where the state is maintained and manipulated through sequential programs written using declarative querying and limited general purpose programming constructs, e.g., stored procedures without modern object-based modularity features. The lack of modularity in this approach makes it hard to identify bugs, isolate failures and maintain application code, especially with growing data and application complexity. Furthermore, the monolithic design makes it hard for the application developer to reason about the scalability and performance of the data tier.

The actor programming model has desirable primitives for concurrency and modularity [3]. By encapsulating state, providing single-threaded semantics for encapsulated state manipulation, and encouraging an asynchronous function-shipping programming paradigm, actors provide a general, modular and concurrent computational model. However, actors expose state management responsibilities such as durability, global state consistency across actors, and failure management to the application developer. Despite these shortcomings, actor frameworks and languages, such as Akka [6], Erlang [13, 56] and especially the virtual actor model of Orleans [23], are increasingly being used to build soft caching layers designed to scale to millions of actors deployed across hundreds of servers using existing cloud-computing infrastructure. These applications vary in diversity, including as examples chat and call services, crowd collaboration and synchronization platforms, mobile and social payment platforms, real-time bidding applications, online games, and even IoT streaming from space balloons [7,57,97]. The popularity of these applications points to the appeal of the actor programming model, which allows application developers to design modular, scalable programs for deployment in an increasingly parallel and heterogenous cloud-computing infrastructure without sacrificing developer productivity.

In line with microservices [69], we believe the data tier should be programmed as a logically distributed runtime with the necessary configurable state management guarantees that are appropriate for the application. In addition, the programming abstraction of the data tier should allow for modular, scalable, performance-portable and cloud-ready design of application programs. Towards this goal, we call on the database community to define and explore the research problems originating from the integration of actor-oriented programming models and classic database management systems.

In this chapter, we propose to center this research agenda around the new abstraction of *actor database systems*, which provides the illusion of a distributed logical runtime enriched with data management features. Actor database systems are envisioned to increase the programmability of the data tier, where the application hard state with strong guarantees of durability and consistency is maintained. As such, actor database systems are complementary to the design of the middle tier, where stateless application logic or alternatively soft-state resides. Even if the middle tier employs actors, as in Orleans [23], such a middle tier cannot subsume the data tier, since only weak state management guarantees are provided and consistency is traded off for availability.

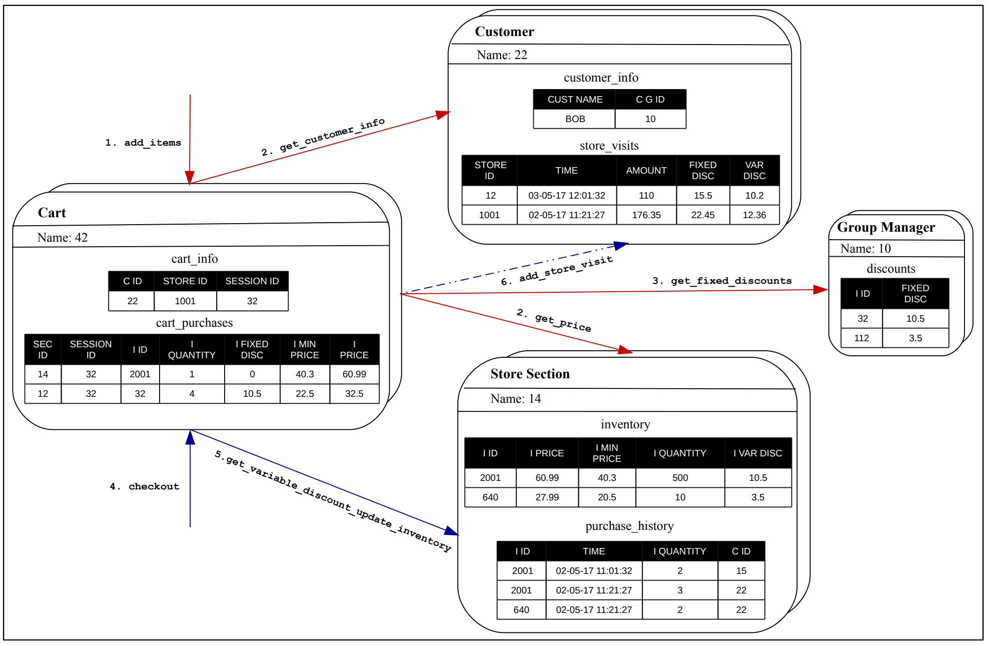
Contributions and Roadmap. This chapter proposes a new vision to marry actors and database systems. Specifically, the chapter makes the following contributions:

1. To motivate the actor database system paradigm, we discuss in Section 2.2 an example of an interactive data-intensive application in the domain of smart supermarkets. The example illustrates at a high level the varied features that a data tier implemented as an actor database system should support.

2. We argue in Section 2.3 why the time is ripe for the emergence of the new abstraction of actor database systems by analyzing a number of current trends in the design of interactive data-intensive applications.

3. Given the aforementioned analysis, we present in Section 2.4 a set of tenets and features that should be followed by every actor database system.

4. To illustrate the potential interplay of these features with applications, we drill down in Section 2.6 on a case study of the smart supermarket application introduced as the running example. After discussing modularity, querying, and transactions through application pseudocode, we illustrate the promise of improved performance with asynchronicity in actor database systems by implementing and evaluating the application in an actor database system prototype.



customer_info

CUST NAME	C G ID
BOB	10

store_visits

STORE ID	TIME	AMOUNT	FIXED DISC	VAR DISC
12	03-05-17 12:01:32	110	15.5	10.2
1001	02-05-17 11:21:27	176.35	22.45	12.36

cart_info

C ID	STORE ID	SESSION ID
22	1001	32

cart_purchases

SEC ID	SESSION ID	I ID	I QUANTITY	I FIXED DISC	I MIN PRICE	I PRICE
14	32	2001	1	0	40.3	60.99
12	32	32	4	10.5	22.5	32.5

discounts

I ID	FIXED DISC
32	10.5
112	3.5

inventory

I ID	I PRICE	I MIN PRICE	I QUANTITY	I VAR DISC
2001	60.99	40.3	500	10.5
640	27.99	20.5	10	3.5

purchase_history

I ID	TIME	I QUANTITY	C ID
2001	02-05-17 11:01:32	2	15
2001	02-05-17 11:21:27	3	22
640	02-05-17 11:21:27	2	22

Figure 2.1: Actors in the data tier of a simplified IoT smart supermarket application. The cart actor supports two transactions, namely add_items and checkout. Several functions are invoked in response to the add_items transaction (1). Both get_customer_info and get_price are asynchronously invoked on the customer and multiple store section actors, respectively (2). Once the customer marketing group is obtained, then get_fixed_discounts is invoked on the group manager actor (3). On checkout (4), get_variable_discount_update_inventory is invoked on each store section asynchronously (5). Finally, a detached transaction add_store_visits is invoked for later execution on the customer actor to record the store visit (6).

In Section 2.8, we discuss related work before concluding.

2.2 Motivating Example

We motivate the integration of actors and database systems by an example that illustrates the needs of many emerging interactive data-intensive applications. Consider a simplified future IoT supermarket application for next-generation self-checkout [11, 103]. The application models the workflow of a customer carrying a smart shopping cart equipped with sensors that can detect physical items inside it. The smart cart periodically interacts with a backend service, which is itself implemented using an actor database system in the data tier. Figure 2.1 shows a set of database actors for this application, along with a chain of function invocations triggered by the functionality to add items and checkout.

The application is functionally decomposed into actors to represent customers, carts, store sections, and group managers for marketing campaigns. For modularity, *state is encapsulated within each actor*; however, for programmability and declarative specification, the state of each actor is abstracted by a set of relations and *application functions employ declarative queries against relations*. For example, the customer actor contains relations recording general information about the customers and their store visits, while the cart actor contains relations recording the contents of the cart. The store section actor contains relations recording the inventory of the store section and its purchase history, and the group manager contains relations recording the fixed discount available on each item specialized for a group of customers.

For database-style consistency in state manipulation, selected *functions perform actions atomically*, in particular the `add_items` and `checkout` functions. When the `add_items` function is invoked, we atomically update the cart with the latest prices and fixed discounts. Similarly, when `checkout` is called, we atomically compute demand-based variable discounts, update inventory, and compute aggregates for the order. In addition, these *functions need isolation under concurrent updates*, e.g., when multiple simultaneous checkouts update the same items in the inventory and calculate demand-based variable discounts. Furthermore, the changes made by `checkout` require *durability* for recording purchase history and updating the inventory.

Even though it may seem natural to treat functions such as `add_items` and `checkout` as classic database transactions, the interaction of database-style functionality with actors brings new challenges and opportunities. In particular, both transactions invoke a number of sub-transactions across various actors. In the actor model, *function calls between actors are asynchronous*. For example, `add_items` asynchronously invokes `get_customer_info` and `get_price` on customer and store section actors respectively, while `checkout` invokes `get_variable_discounts_update_inventory` on store section actors. This asynchronicity in invocations exposes *intra-transaction parallelism*, which must interplay cleanly with transactional semantics. Moreover, we may not always wish that asynchronous function calls be part of the same transactional context. For example, `checkout` triggers a *detached transaction* `add_store_visit`, which is separately executed at a later time, to record a trace of store visits for the customer. In such a case, *flexibility in fault tolerance guarantees* can be afforded to the detached transaction for recording the purchase of the customer, since independent failure of a customer actor does not preclude the main application functionality from being executed. Similarly, `add_items` might not need durability since it can be recomputed from the cart's physical contents albeit with different prices and fixed discounts (if inventory or group manager prices and discounts change) depending on application semantics. Finally, asynchronicity also has implications on declarative querying functionality. While queries to the state of an actor are synchronous, the semantics of multi-actor queries involving asynchronous function calls needs to be carefully considered.

2.3 Why Actor Database Systems Now?

In this section, we outline the technological and application design trends that act as key enablers for the use of actor database systems.

2.3.1 Popularity of Cloud Computing Platforms, Middleware and Microservices

The last decade has witnessed a massive growth in the amount and variety of web services, which have targeted cloud computing infrastructure for deployment. Utilizing a three-tier architecture, these services employ a stateful middle tier using web-application frameworks or language runtimes, while the data tier consists of a database system. Web-application frameworks employing asynchronous and reactive programming, actor runtimes and NoSQL data stores have seen widespread adoption for constructing these interactive web services. Increasingly, application logic has been moved away from the database system into the middle tier, repurposing the database system as a fault-tolerant, consistent storage layer. Originally, this migration of logic to the middle tier was a response to scalability and performance concerns of the data tier, but of late programming flexibility and development productivity have emerged as major drivers [102, 118, 119].

However, this approach has not been without its share of failings. The lack and misunderstanding of data consistency semantics, fault-tolerance models, and query capabilities in the middle tier have affected application correctness [18, 46], which has raised voices for integrating database features into middle tier platforms [26]. The growth of microservices as an architectural pattern has made a case for a modular, scalable, fault-tolerant design of these web services to avoid the pitfalls of a monolithic architecture [69]. However, current deployments of microservices argue for containerization of whole software components, which raises the operational cost of these services and burdens the application to administer and integrate these modular software systems.

Actor database systems have the potential to address all these existing gaps. They provide a concrete programming paradigm to functionally decompose the data tier in modules across actors in line with the microservice architecture, but at a logical level independent of actual software system components and without paying a high overhead for modularity and encapsulation. They also incorporate well-understood, decade-strong data management and fault-tolerance guarantees, which relieves developers from the burden of reasoning about complicated state consistency semantics under asynchronous, concurrent executions of application programs in the presence of failures. In short, actor database systems increase the programmability of the data tier, which can then be conceived as a language runtime with robust state management features instead of just a storage abstraction.

2.3.2 Modular, Elastic, Available and Heterogeneous Applications

The popularity of microservices points to the importance of a modular design as a solution to manage application complexity and failures. A modular design helps in better fault isolation, debugging and profiling of the individual modules as opposed to a monolithic design, thus increasing programming productivity. In addition, a modular design can improve availability by a fail-soft strategy, where faulty modules and its dependents can be made unavailable instead of an entire application. A modular design also aids in targeted monitoring of the modules and better analysis of impacts, which can bring substantial benefits in load provisioning and resource utilization with workload changes. The latter has a direct impact on supporting elasticity for an application, which has become important today given the 24x7 nature of online web services and the changes in workload they go through.

Modularity is also an important building block in supporting the heterogeneous requirements of current web services. For example, an application might have varying durability needs where (1) the entire data need not be durable and/or (2) only executions of certain programs need to be durable. Another case can be made for encrypted data where the entire data need not be encrypted and engender the associated overheads. Similarly, a case can be made for concurrency control, where different subsets of data and programs can benefit from different concurrency control structures and isolation levels. Currently, these heterogeneous needs are explicitly managed by the application by deploying associated software components, which significantly increases application complexity and maintenance overheads.

Actor database systems with their modular, concurrent and fault-tolerant programming model can cater to these application needs in a manner that is both more natural and better integrated with data management functionality. By allowing decomposition across actors and supporting actor heterogeneity, actor database systems can allow application programmers to declare the durability, encryption and concurrency needs of each actor, thus making such actors the islands of homogeneity in the application.

2.3.3 Increasing Parallel Hardware

Over the past two decades, computing power has increased dramatically. Initially, this increase came in the form of higher clock rates for processors; of late, in the form of more processing elements (cores) in a single chip. The cost of these computing elements has gone down dramatically as well, making them widely affordable. Dropping costs and improving performance have also been witnessed in storage and networking technologies, which has given rise to new challenges for database systems to transition these hardware benefits to applications [10]. Even though database systems lie at the cross-section of system software (OS) and application programs, they lack abstractions to expose the available physical parallelism using a high-level programming model.

To this end, actor programming models hold promise to fill this gap, as they possess: (1) well-defined concurrency and asynchronous message passing semantics, and (2) a function-shipping programming paradigm. Actors allow for portable specification of applications in terms of high-level, application-defined concurrent computational elements independent of the actual physical hardware and operating system primitives. With an actor-based specification, the available control-flow parallelism in application functions can then be leveraged from a higher level of abstraction to improve application performance. Additionally, actor database systems can exploit the locality information encoded in actors to better target classic data management optimizations, e.g., for index structure layouts, code generation, and transaction affinity.

2.3.4 Latency Sensitive Applications

The last few years have witnessed a rapid growth of stateful, scalable, latency-sensitive web services in various application domains, e.g., online games, mobile and social payment platforms, financial trading systems, and IoT edge analytics [102, 118]. The increasing adoption of scalable actor runtimes such as Akka, Erlang and Orleans to deploy these services points towards the attractiveness of actor programming models for designing these applications [119]. By providing single-threaded execution with asynchronous message passing semantics, actors simplify concurrent programming while enabling developers to leverage available asynchronicity in the design of latency-sensitive and locality-aware applications.

However, actor runtimes shift the responsibility for state management under failures to the programmer, which has raised the need for integrating classical database state management functionality [26]. The data model in these runtimes is low-level, language-dependent and lacks declarative query capabilities, thus pushing physical design decisions into applications.

Actor database systems have the potential to simplify application development and portability without compromising programming flexibility and correctness in this growing space of stateful, latency-sensitive applications. To this end, actor database systems provide robust state management guarantees as well as high-level data model and query capabilities, bringing physical data independence to the actor programming model.

2.3.5 Security Risks

With increasing pervasiveness of software services, security challenges faced by these services continue to grow dramatically. These challenges include issues pertaining to data integrity, access control, authorization, monitoring and auditing of these services. It is equally important to detect security violations and to mitigate them with minimum possible impact on the service operation. With increasing size of application deployments and complexity, it is imperative that software tools support specification of secure application code and help in static and dynamic verification.

Actor database systems open up new possibilities to re-think the database security model for current and future applications. For example, the traditional security model based on users and roles can be augmented in actor database systems with object-capability security, aiding in monitoring information flow on message passing. Having a modular architecture can also enable auditing of security violations and upon incidents, help in limiting unavailability to only the affected actors and/or functions instead of all actors and all functions.

2.4 Actor Database System Tenets

In this section, we outline the tenets that identify an actor database system based on the analysis in Section 2.3. We further enumerate and classify the actor database system features under these basic tenets. We propose mandatory features that we envision a system must support to qualify as an actor database system and meet the design trends of interactive data-intensive applications. However, we do not consider this feature set as final, but as a concrete formulation for further discussion in the community. We additionally list a set of optional features that were part of our reflections in Section 2.5.

2.4.1 Overview

Tenet 1: Modularity and Encapsulation by a Logical Actor Construct. In order to tackle growing application complexity, isolate faults, define heterogeneous application requirements and increase programming productivity, modular programming constructs providing encapsulation are required. It is also desirable that the overhead of modularity be as low as possible and that modules be defined by the application independently of the hardware and software used for deployment. Actors provide this low-overhead computational construct in an actor database system as opposed to objects, which are a data encapsulation mechanism only.

Tenet 2: Asynchronous, Nested Function Shipping. For latency-sensitive applications to leverage the benefits of increasingly parallel hardware, asynchronicity in communication among actors becomes necessary. By using asynchronous function shipping to communicate with an actor and by utilizing nested invocations of such asynchronous functions, an application can leverage the available application control-flow parallelism to minimize latency, increase locality of data accesses in functions, and thus improve application performance using high-level programming abstractions.

Tenet 3: Transaction and Declarative Querying Functionality. In order to ease the burden of managing complexity and to increase programming productivity, a well-defined model for concurrency and fault-tolerance is desirable. Single-threaded execution semantics in actor models has been extremely appealing to programmers, hinting at the potential of borrowing and adapting classic database mechanisms such as transactions and more specifically nested transaction models [129]. Moreover, to bring physical data independence to actor models, a high-level data model and declarative query capabilities are required for easy interaction with encapsulated state within the actors.

Tenet 4: Security, Monitoring, Administration and Auditability. In order to address security threats, support for limiting information flow is required from the programming model during design and construction of applications. In addition, at runtime, monitoring, administration and auditing components provide for further manageability of security violations. By virtue of modularity and encapsulation in actors, a host of language-based security techniques can be provided to enrich the programming model to reason about security of the application during the design phase and to enable software verification. Actor modularity also enables targeted monitoring, auditing, administration and response to security threats in order to minimize availability issues.

2.4.2 Tenet 1: Modularity and Encapsulation by a Logical Actor Construct

Mandatory Feature 1: *Logical, Concurrent, Distributed Actors With Location Transparency.* For modularity, an actor database system must provide a programming abstraction of concurrent and distributed logical actors. A logical actor is an application-defined processing entity that communicates using message passing. Messages can be modeled as a computation that causes a response (see Tenet 2). Logical actors are concurrent and distributed because every logical actor is isolated from each other and can run at the same time independently. A logical actor does not imply a one-to-one mapping to an actual physical element (e.g., thread or process) used to implement it, but is merely an application modeling construct.

Even though logical actors communicate via message passing, we define a logical actor to *not* have a mailbox abstraction as a traditional actor [4]. Hence, a logical actor must not be conceptualized as an interrupt handler where the application developer needs to implement the message receiving loop. Rather, a logical actor is a reactive entity that processes requests submitted to it from clients or other logical actors. The decision to make logical actors reactive does not preclude the use of application-defined mailboxes, but does not force this design onto every application over an actor database system.

The application developer can model and structure her application with logical actors as in Figure 2.1. Note that the relations displayed in the figure could have been alternatively grouped under a different actor design. Consequently, a given application can be structured in multiple ways depending on how logical actors are defined. We envision that a logical actor would be understood by developers as a logical thread of control and used to capture the units of scaling and available parallelism of the application. For example, the whole set of logical actors in Figure 2.1 could be instantiated once per store or group of stores. At finer granularity, the application could be scaled on the number of carts and sections within a store.

A logical actor primitive can be provided in multiple ways. An object-oriented abstraction [23, 34] can be used to provide a logical actor primitive where the methods of the object define the computations that can be invoked on the logical actor. A function-oriented primitive can also be used by just declaring the logical actors and then defining computations in terms of them. In contrast to objects in object-oriented database systems [15], however, actors in actor database systems are not simply an abstraction to encapsulate data and define behavior on it. Logical actors provide the illusion of a thread of control and thus allow the developer to explicitly model the scalable units of her application as well as asynchronous computation and communication as discussed in Tenet 2. Logical actors allow decomposition of the data tier in a modular fashion allowing for better isolation of bugs, containment of failures, and management of application complexity.

Every logical actor must be uniquely identified by a name. In other words, a logical actor must be assigned a name from a logical namespace by the application. The name of a logical actor acts as the logical actor's sole identity from a programming perspective. For example in Figure 2.1, the name of a customer actor is the implicit primary key of the `customer_info` relation, i.e., the customer ID.

Mandatory Feature 2: *Actor Lifetime Management.* Since the programming model exposes primitives for specifying application-defined logical actors, a natural question is whether the application needs to manage the life cycle of these actors. Traditionally, actor models have exposed actor lifetime management to the application. However, an actor database system can either choose to manage actor lifetime itself or delegate it to the application by selecting any of the following mechanisms:

(1) *Dynamic Actor Creation*: Similarly to dynamic memory allocation, the application can be given the control to create and destroy logical actors dynamically. An attempt to create a duplicate logical actor or to destroy a non-existent logical actor must be signaled by appropriate errors. This policy is similar to the model supported by existing actor language runtimes, such as Akka [6] or Erlang [13,56].

(2) *Static Actor Creation*: An actor database system can also choose to manage actor lifetime itself and therefore not provide primitives for actor creation and deletion. In this mechanism, the actor database system creates the illusion of logical actors to be in perpetual existence to the application. This can be supported by (a) *automatic actor creation* [34], where accessing an actor by name automatically creates that actor; or (b) *declarative actor creation*, where the application declares the names of the logical actors that should be available for the lifetime of the application.

2.4.3 Tenet 2: Asynchronous, Nested Function Shipping

Mandatory Feature 3: *Asynchronous Operation Support.* Asynchronous messaging is necessary as a communication mechanism for logical actors to provide a programming construct for the application developer to reason about control and data flow dependencies and explicitly expose parallelism in a computation for performance. Asynchronous messaging across actors can be implemented in various ways, e.g., using traditional actor model message passing [3], or method invocations on objects returning promises [34], to name a few. Irrespective of implementation mechanism, asynchronous logical actor messaging must allow the caller to synchronize on the result of the communication if desired, i.e., if the communication is a send primitive then the client can choose to wait until the message is successfully received; similarly, if the communication is through a remote procedure call, then the client can choose to wait until its result is propagated back. For example in Figure 2.1, the functions `get_customer_info` and `get_price` are invoked asynchronously within the `add_items` transaction. Similarly, multiple calls to `get_variable_discount_update_inventory` are invoked asynchronously within `checkout`.

By introducing asynchronicity in the messaging mechanism, the illusion of a purely sequential program is broken. Racy computations may now be possible, i.e., in the absence of any concurrent computation and given the same input state, the program execution can produce inconsistent result states depending on the order in which asynchronous computations are scheduled on a conflicting data item. Either the programming model must disallow statically the formulation of such programs or the runtime must reject such malformed programs. In addition, the impact of asynchronous messaging on the isolation semantics exposed by the programming model of an actor database system must be clearly defined.

2.4.4 Tenet 3: Transaction and Declarative Querying Functionality

Mandatory Feature 4: *Memory Consistency Model.* An actor database system must provide primitives to allow an application to control the isolation level across concurrent computations. Although enforcing before-or-after atomicity (serializability) of computations on the union of the states of all actors in an actor database would be the simplest isolation level to program with in order to guarantee correctness, evidence suggests that applications can tolerate lower isolation levels with alternative guarantees and mechanisms [18, 109]. At the same time, and in contrast to microservice architectures [69], actor database systems should not shy away from defining consistency semantics for global state manipulation across multiple actors.

Isolation levels and their control can be provided in various ways: (1) Borrowing traditional database isolation levels and exposing them as annotations in actor computations; (2) Following a turn-based model similar to Orleans [23]; (3) Employing application-defined invariant-based isolation guarantees [17, 109]. In all of these alternatives, isolation in sub-computations of a given computation must be considered carefully. For example, the isolation level of a parent computation could be propagated to child computations, the child could remain independent of the parent, or isolation level for the parent and the child computations should be compatible.

Mandatory Feature 5: *Fault-Tolerant Actors With Application-Defined Durability.* An actor database system must free the application developer from worrying about partial alteration of actor state by an incomplete computation under failure. Consequently, an actor database system must support the classic notion of all-or-nothing atomicity of computations (recoverability). Still, detached transactions running independently from a calling transaction should be provided to allow for flexibility in fault-tolerance guarantees and performance. For example, the logic of the `add_store_visit` call in Figure 2.1 is executed as a detached transaction with respect to `checkout`. To establish a fault-tolerance contract in the call, an exactly-once qualifier could be added to the `add_store_visit` invocation. Different detached transaction invocations may have different qualifiers, e.g., for at-most-once or at-least-once semantics.

In addition to recoverability, database systems guarantee durability of committed transactions. By contrast, actor runtimes do not provide any durability guarantees for computations, forcing the application developer to store either parts or the entirety of actor state in an external storage system. To allow flexibility of application design, the programming model of an actor database system must fall in-between the two extremes and allow an application to control actor state durability. The logical programming model must thus support the notion of durability as a property to avoid conflation with physical deployment as is the case in actor systems. For example in Figure 2.1, the application may choose *not* to make the cart actor durable, since it can always be reconstructed if needed by reading the contents in the physical shopping cart itself. Alternatively, durability annotations could be specified per computation and not per actor. In contrast to early systems such as Argus [87], however, such mechanisms must work in conjunction with a high-level data model and declarative querying, as discussed in the following features.

Mandatory Feature 6: *High-Level Data Model and Declarative Querying Support.* The state of a logical actor must be abstracted by a high-level data model to provide for physical data independence. An application developer should have full freedom in defining the schema of a logical actor fitting the application needs, thus allowing schema definitions to vary across logical actors. This allows the application in Figure 2.1 to specify appropriate schemas in the relational model for the different actors, and frees the application developers from worrying about the physical data layout. The actor database system must also provide a declarative query facility over the state encapsulated in a single actor. This query facility provides ease of programming and allows for reuse of existing database query optimization machinery for performance.

Mandatory Feature 7: *Multi-Actor Declarative Querying.* Declarative querying in an actor database system must extend to multiple actors. Multi-actor query support could be added by prepending actor names before relations, objects, or other data model constructs similarly to object-oriented query languages [8, 85]. However, this simplicity is elusive: as opposed to object models, the actor model includes explicit asynchronicity in any multi-actor accesses, which must happen through message passing or function calls. Thus, the semantics of asynchronous execution of computations needs to be reconciled with the simplicity of a declarative interface. In addition, query optimization needs to be revisited to take into account asynchronicity elements in query specifications.

2.4.5 Tenet 4: Security, Monitoring, Administration and Auditability

Mandatory Feature 8: *Actor-Oriented Access Control.* Actor encapsulation and modularity provide security by ensuring that an actor's state can only be accessed through its methods and by localizing security breaches. This enables standard static verification of illegal accesses by information flow analysis. However, this mechanism can be enriched further with access control features found in classic RDBMS, in particular fine-grained access control models [77, 107]. Such an integration would allow, for example, rich access specifications by methods of actor types and/or particular actor names to other methods of actor types and/or given actor names. By allowing both static and dynamic configuration of access control, static verification and debugging utilities can enrich the design process while dynamic changes protect against violations at runtime. In addition, to further protect against unauthorized access, actors should allow specification of encrypted relations in their state and annotation of methods as encrypted to ensure all communication to and from the methods are encrypted as well.

Mandatory Feature 9: *Administration and Monitoring.* An actor database system should also provide an administrative interface for flexible maintenance, allowing addition and removal of actors, changing resources allocated to them and modifying access control specifications. Furthermore, an actor database system must support targeted monitoring of actors by gathering statistics of actor usage as well as audit traces of actor method executions, potential security violations, and anomalous accesses. Taken together, this functionality enables administrators to intervene in the system at a fine granularity, e.g., removing or deactivating specific actors that are detected as exhibiting malicious behavior.

2.5 Optional Features for Actor Database Systems

In this section, we list a number of features for actor database systems that we have classified as optional. The integration of these features promises interesting research challenges, but the need for their support based on interactive data-intensive application design trends is not as clear as for mandatory features. We refrain from classifying the optional features under specific tenets, since some of the features may have interactions with multiple tenets.

Optional Feature 1: *Actor Computational Heterogeneity.* The traditional actor model and its supported platforms advocate a notion of computational homogeneity, i.e., every logical actor has equal processing power [4]. However, in certain applications, some actors may have to sustain higher computation load than others. For example in Figure 2.1, a store section might have to sustain a higher load than a customer actor.

An actor database system can support programming constructs to allow the application developer to declare computational heterogeneity dependencies across actors, e.g., $ActorX > 2 \times ActorY$. These declarative specifications can be extended to express memory and communication requirements across actors as well. These logical actor computational relationships can then be leveraged in the actual deployment of logical actors on physical hardware to adapt to application load and to better specify and sustain application service level agreements.

Optional Feature 2: *Data Model Heterogeneity.* Given the growth in variety of data-intensive applications, it would be hard to presume that a single data model across all logical actors would suffice for all application needs. In order to provide more flexibility, an actor database system could allow disjoint parts of the state in and across actors to be abstracted by different data models (e.g., relational, object oriented, XML). Such data model heterogeneity would allow for rich modeling of state within and across logical actors, avoiding re-architecting an application on top of a single data model. In contrast to recent proposals for polystores [53], however, an actor database system must provide the mandatory features discussed in Section 2.4. For example in Figure 2.1, the inventory and purchase history could be stored using an XML or object-oriented data model in the store section actor.

Optional Feature 3: *Language Integration and Computational Completeness.* One of the reasons for the popularity of NoSQL and MapReduce systems has been their mature integration with high-level programming languages, which makes it very productive for application developers to interface their application code with such systems. Therefore, tighter integration of programming languages in an actor database system is a desirable property. Since actor database systems aim at a large set of interactive data-intensive applications, an actor database system with a strong programming language integration providing computational completeness has a stronger appeal for adoption than one exposing a computationally incomplete domain-specific language embedding. Moreover, easy integration of the actor database system infrastructure with multiple target language dialects would increase the chances of adoption even further.

2.6 Case Study: SmartMart

In this section, we perform a case study of the smart supermarket application outlined in Figure 2.1 and briefly described in Section 2.2 in light of the enunciated tenets and the feature set in Section 2.4. We revisit each of these tenets and features to provide a concrete example of each in the context of this application.

2.6.1 Application Logic Overview

The *SmartMart* application models interactions in a supermarket where carts are equipped with sensors to read the physical cart contents and to trigger checkout operations. These sensors periodically interact with the back-end service in the data tier to call `add_items` and `checkout`. In a real application, more operations, such as to remove items from the cart, need to be supported, but we omit these additional operations for brevity.

In the application, the discount available on an item is classified into two components: (1) fixed discount and (2) variable discount. The fixed discount is customized for a marketing group of customers, while the variable discount is computed based on the demand for the item over a predefined window. Every item has a minimum price as well to ensure that discounts do not overshoot it. *The price and fixed discount are computed when items are added to the cart, while the variable discount is only computed at checkout.* For an item i, if $q_{i,t}$ represents the quantity bought at time t and $S^k_{i,t}$ represents the set of quantities from the reverse purchase history of the item starting at $t-1$ and size at most k, then the variable discount is computed using the following formula:

$$vdisc_{i,t} = \frac{q_{i,t}}{\mu(S^k_{i,t}) \quad + \quad c \times \sigma(S^k_{i,t})} \times VD_i \tag{2.1}$$

For a tunable and predefined constant c, the denominator in the fraction models a target purchase quantity from the history of purchases based on the mean plus c standard deviations of the purchase quantity distribution. VD_i is the predefined variable discount that such a purchase would receive. Thus, the current purchase is normalized by the target purchase and then multiplied with VD_i to compute the dynamic variable discount.

2.6.2 Tenet 1

According to **Feature 1**, an actor database system must provide a construct to create logical actors. In Figure 2.3, the keyword `actor` is used to specify two application-defined actor types, namely (1) `Customer` and (2) `Group_Manager`. An actor type definition must also include the encapsulated state using the supported data model, which is bound to the lifetime of the actor. In the example, the state has been abstracted using relations, whose schema definition is omitted for brevity.[1] Each actor type also defines the set of methods that can be invoked on the actor. In contrast to methods in classic object-based models, actor models explicitly define that method calls must be logically shipped for execution on the desired actor, since each actor is a logically concurrent entity. Furthermore, the state encapsulated in each actor can only be accessed by invoking methods defined by the actor. An actor method can contain any sequence of queries, procedural logic and invocations to

[1] The full pseudocode for the example is available in Section 2.6.6.4. For all code examples, we make simplifications in type annotations and conversions to keep the pseudocode brief.

```
  CREATE ACTORS OF TYPE Customer WITH NAMES IN (22, 32);
  CREATE ACTORS OF TYPE Cart WITH NAMES IN (42, 43);

  DROP ACTORS OF TYPE Cart WITH NAMES IN (42);
```

Figure 2.2: Example of Actor Creation and Removal.

```
actor Customer {
   state:
     relation customer_info (...);

     relation store_visits (...);

   method:
     tuple get_customer_info() {...};

     void add_store_visit(int store_id, timestamp time,
                          float amt, float fixed_disc,
                          float var_disc) {...};
};

actor Group_Manager {
   state:
     relation discounts (...);

   method:
     list<tuple> get_fixed_discounts(list<int> items) {...};
};
```

Figure 2.3: Actor definitions.

other actors. Any variable defined in actor methods and not in the state has the lifetime of the method only and not the actor.

In line with **Feature 2**, an actor database system must provide automatic actor lifetime management. For example, in a system supporting static declarative actor creation, commands to create and delete actors should be provided to support actor creation and removal. Figure 2.2 is a sample script which illustrates how to create and delete actors with the appropriate type and name:[2]

The declared actors are available for the lifetime of the application from the point of their creation to the point of their deletion, and the actor database system is responsible for managing the lifecycle of these actors. Any invocation of a method on an actor not created would result in an error. Moreover, the use of these commands are for administration only and are disallowed from the method body of the actors as per the static actor creation policy.

[2] Integer values are used as actor names in the SmartMart application for simplicity; in general, an actor name can be any application-defined string.

2.6.3 Tenet 2

According to **Feature 3**, an actor database system must support nested asynchronous function shipping. To exemplify this feature, we abstract method invocations on actors as asynchronous function calls returning *futures*, which represent the result of the computation. In Figure 2.4, we illustrate how method invocations and result synchronization could be operationalized in an application program. The figure shows the pseudocode of the `add_items` method in the `Cart` actor. For simplicity, we assume that the cart is private for a customer, and the method is only invoked once for all the items ordered in the cart before a checkout is performed. Within the pseudocode of `add_items`, further method invocations to other actors are performed. An invocation of a method on an actor must specify the type of the actor within the <> brackets and the name of the actor within the [] brackets, followed by the method and its arguments. For example, in the first loop in the program, we invoke the `get_price` method on each of the store sections in the item orders. The call is directed to an actor of type `Store_Section`, whose name is given by the section ID. The method gets as argument a list of the items across all item orders for the corresponding section ID. As a result, the method call produces a future. All futures are collected in a map for synchronization at a later time. As such, the subsequent logic in `add_items` is executed while the asynchronous calls to `get_price` are processed.

The pseudocode employs an imperative style of invocation of asynchronous function calls in different actors, demonstrating the flexibility in actor models in encoding arbitrary control flow logic and dependency in application code. For example, after the first loop, an invocation to the customer actor to get the customer group is made, and the future is immediately synchronized upon since the customer group value is necessary for invocation of `get_fixed_discounts` on the group manager actor. So the computation of `get_price` may overlap with both `get_customer_info` and `get_fixed_discounts`, while `get_customer_info` is sequential wrt. `get_fixed_discounts`. Furthermore, a method invocation on an actor can trigger further asynchronous method invocations to other actors, thus allowing arbitrary nesting. In such a case, a method invocation only completes when all its nested method invocations complete. Actors support function shipping by design, since methods can only execute on the actor that defines them and hence the notion of locality is implicit.

Different schemes can be used to synchronize on future results. The calling code can synchronize on the future by invoking *get* when the value is needed. For example, this is done to obtain the result value from the `get_fixed_discounts` method call. Alternatively, multiple futures can be synchronized upon at the same time by calls such as `when_all` or `when_one` to consume result values when all or any one are available, respectively. In the example, the futures from the price lookups are synchronized upon using the barrier semantics (*when_all*), after which cart purchases are recorded and an updated session ID value is returned for later use during checkout.

```
nondurable actor Cart {
   state:
     ...

   method:

    int add_items(list<order> orders, int o_c_id) {
      ... // Organize the item ids in orders by store section

      map<int,future> results;
      for (section_order : orders_by_store_section) {
        future res := actor<Store_Section>[section_order.sec_id].
                           get_price(section_order.item_ids);
        results.add(section_order.sec_id, res);
      }

      future c_g_res :=
                   actor<Customer>[o_c_id].get_customer_info();
      int c_g = c_g_res.get();

      ... // Compute list of all ids of ordered items

      future disc_res := actor<Group_manager>[c_g].
                              get_fixed_discounts(ordered_items);

      ... // Generate session_id and update cart_info

      list<tuple> discounts = disc_res.get();

      results.value_list().when_all();

      ... // Iterate over prices and discounts and store in cart_purchases

      return v_session_id;
    }

    ...
};
```

Figure 2.4: Implementation of `add_items` in the `Cart` actor.

2.6.4 Tenet 3

Feature 4 specifies that a memory consistency model must be defined by an actor database system to clarify the semantics of multiple method invocations on the same actor and across actors. While multiple memory consistency models are possible, in our example we adopt classic database serializability, and propagate this same isolation level across all nested method invocations. This simplifies the application code in Figures 2.4 and 2.5, since the application developer is insulated from concurrent and asynchronous manipulation of actor state within and across multiple transactions. Furthermore, in order to maintain well-defined results under asynchronicity, any two conflicting sub-computations on the same actor must be ordered by the application code via synchronization using futures; otherwise, the actor database system aborts the entire computation.

In line with **Feature 5**, we adopt all-or-nothing atomicity of methods and application-defined durability per actor in our example. All-or-nothing atomicity frees the developer from worrying about partial state changes encapsulated by one or many actors, and thus frees the application logic from implementing failure-handling code as in Figures 2.4 and 2.5. Application-defined durability per actor allows flexible specification of durability requirements. In the example, the `Cart` actor is annotated as `nondurable`, while the other actors are durable by default. So, all state manipulation of committed transactions is durable on the `Customer`, `Group_Manager` and `Store_Section` actors only. This allows the application to flag cart state as transient, while not giving up all other data management features in implementing cart operations.

In addition to durability annotations, the notion of *detached transactions* allows for invocations of sub-computations in a separate transactional context, i.e., the sub-computation does not share the isolation level and atomic commitment requirements of the caller [100]. The caller can specify when the detached computation should be invoked, e.g., on successful commit of the caller, abort of the caller, or any of them. This feature has been used in Figure 2.5 to invoke a detached transaction on the customer actor so as to record purchase information on successful commit of `checkout`.

According to **Feature 6**, a high-level data model and declarative state querying capabilities must be provided. In Figure 2.5, `cart_info` exemplifies a relation schema abstracting portion of the encapsulated state of the actor type `Cart`. The method `checkout` interacts with the encapsulated `cart_info` and `cart_purchases` relations using declarative queries in SQL.

Figure 2.5 also exemplifies **Feature 7**, since declarative multi-actor querying is employed in contrast to the use of imperative constructs and explicit future synchronization in application code in Figure 2.4. The figure shows a SQL query that invokes `get_variable_discount_update_inventory` in all store sections that have participated in the given session. The list of items passed as input to each invocation is constructed by converting the relational result of a nested query to a list by function `LIST`. The result of the invocations is a relation with price and discount information per store section. This result is aggregated in the top-level SQL query to compute the total amount bought along with total fixed and variable discounts for the checkout.

A number of issues need to be considered when specifying the semantics of such multi-actor queries. First, the calls to `get_variable_discount_update_inventory` are asynchronous, but the query needs the actual values returned from the calls in its execution. This suggests a semantics in which the values are available immediately, as if the calls were synchronous, but that allows for the query optimizer to delay future evaluation as much as possible. Second, since calls are dispatched in bulk in SQL, it may be necessary for application logic to differentiate the results coming from different actors. In the query, we assume any values returned from an asynchronous function invocation can be converted to sets of tuples

```
nondurable actor Cart {
  state:
    relation cart_info (c_id int, store_id int,
                        session_id int);

    relation cart_purchases (...);
  ...
  method:

    float checkout(int c_session_id) {
      SELECT * INTO v_cart FROM cart_info;
      timestamp v_c_time := current_time();

      SELECT SUM(amount) amt, SUM(fixed_disc) fixed_disc,
             SUM(var_disc) var_disc
      FROM (SELECT (SELECT amount, fixed_disc, var_disc
                    FROM actor<Store_Section>.
                         get_variable_discount_update_inventory(
                             v_cart.c_id, v_c_time,
                             LIST(SELECT i_id, i_quantity,
                                         i_price, i_fixed_disc,
                                         i_min_price
                                  FROM   cart_purchases
                                  WHERE  sec_id = S.sec_id
                                   AND   session_id =
                                         S.session_id))
                    WHERE name = S.sec_id)
            FROM (SELECT DISTINCT sec_id
                  FROM cart_purchases
                  WHERE session_id = c_session_id) S);


      DETACH actor<Customer>[v_cart.c_id].add_store_visit(
                 v_cart.store_id,v_c_time,amt,fixed_disc,var_disc)
      ON COMMIT;

      return amt;
    }
    ...
};
```

Figure 2.5: Implementation of `checkout` in the `Cart` actor.

with the actor name included as a column. In this way, we can select the right actor from a set of results through predicates such as the WHERE clause name = S.sec_id. Third, asynchronous function calls may update state, as is the case with get_variable_discount_update_inventory. Actors have disjoint state, so from the query it is possible to determine statically that the updates will not be conflicting given that each invocation goes to a distinct actor and there are no further nested calls. However, the restrictions on updates in asynchronous function invocations embedded in multi-actor queries need to be carefully thought out, e.g., to ensure commutativity and associativity of operations.

2.6.5 Tenet 4

By **Feature 8**, actor-oriented access control should enrich traditional object-based access modifiers with fine-grained access control as studied in database systems. In the SmartMart example, suppose we wish to configure minimum levels of access to mitigate security risks so that: (Rule 1) add_items in Cart actors has access to get_price in Store_Section actors, get_fixed_discounts in Group_Manager actors and get_customer_info in Customer actors; (Rule 2) checkout in Cart actors has access to get_variable_discount_update_inventory in Store_Section actors and add_store_visit in Customer actors; and (Rule 3) A set of specific cart instances (e.g., carts 12, 13, and 14) can only interact with the sections of their corresponding physical store (e.g., store sections 100 and 200). This access configuration can be enforced in an actor database system using the commands listed in Figure 2.6:

The first *REVOKE* statement revokes access rights of all actors to each other. The next *GRANT* statement configures Rule 1 for access privileges of add_items, while the next statement configures Rule 2 for access privileges of checkout in the Cart actor type. The final statement additionally sets up a rule by actor name to enforce Rule 3. The previous rules, which are configured by actor types, can additionally use the WITH NAMES clause to configure even finer granularity of access. Taken as a whole, the set of rules must cleanly compose or otherwise be flagged and rejected. The set of configured rules can be used for static verification and debugging of security violations. In addition, modification of these rules during deployment enables dynamic adaptivity of security policies to meet unforeseen security threats, e.g., by revoking rights from selected actors.

Since the application functionality is deconstructed in terms of actors, **Feature 9** implies that the actor database system should provide monitoring of actor usage and resource utilization (e.g., some store sections being more loaded than others), security violations (e.g., a Group_Manager actor attempting to access a Store_Section actor), and audit traces (e.g., traces of checkout and nested method invocations). It also implies administrative support to scale actors and resources to meet usage fluctuations discovered during monitoring, or change access control specifications based on security violations, to name a few possibilities.

```
REVOKE ACCESS TO ACTORS OF TYPE ALL FROM ACTORS OF TYPE ALL;

GRANT ACTORS OF TYPE Cart WITH METHODS IN (add_items)
 ACCESS TO
   ACTORS OF TYPE Store_Section WITH METHODS IN (get_price)
 AND ACCESS TO
   ACTORS OF TYPE Customer WITH METHODS IN (get_customer_info)
 AND ACCESS TO
   ACTORS OF TYPE Group_Manager WITH METHODS IN
                  (get_fixed_discounts);

GRANT ACTORS OF TYPE Cart WITH METHODS IN (checkout)
 ACCESS TO
   ACTORS OF TYPE Store_Section WITH METHODS IN
                  (get_variable_discount_update_inventory)
 AND ACCESS TO
   ACTORS OF TYPE Customer WITH METHODS IN (add_store_visit);

GRANT ACTORS OF TYPE Cart WITH NAMES IN (12,13,14)
 ACCESS TO
   ACTORS OF TYPE Store_Section WITH NAMES IN (100, 200);
```

Figure 2.6: Configuration of fine-grained access-control of `Cart` actors in SmartMart

2.6.6 Evaluation: Asynchronicity + Transactions

In this section, we evaluate: (1) The potential for performance gains provided by asynchronous communication in actor databases (Section 2.6.6.2); and (2) The effect of load under concurrency on transactions with asynchronicity (Section 2.6.6.3).

2.6.6.1 Experimental Setup

We present the hardware, workload, system prototype, and methodology used for the evaluation.

Hardware. We employ a machine with two sockets, each with one eight-core 2.6 GHz Intel Xeon E5-2650 v2 processor with two physical threads per core, leading to a total of 32 hardware threads. Each physical core has a private 32 KB L1 data and instruction cache and a private 256 KB L2 cache. All the cores on the same socket share a last-level L3 cache of 20 MB. The machine has 128 GB of RAM in total, with half the memory attached to each of the two sockets, and runs 64-bit RHEL Linux 3.10.0.

Workload. We used the SmartMart application for our experiments. To simulate the workings of one store, we created eight `Store_Section` actors. For each, we loaded the `inventory` relation with 10,000 items and the `purchase_history` relation with 300 entries per item for a total of 3,000,000 entries, simulating a history of 120 days where 500 customers on average visit the store per day and buy 50 items each. We fix the number of `Group_Manager` actors to 10 and vary the number of `Cart` actors depending on the experiment. The number of `Customer` actors is set to 30 times the number of carts. To calculate the variable discount, we tuned the window size to correspond to 150 records in the `purchase_history` relation, thus calculating a target purchase quantity over 60 days (Equation 2.1). The entire size allocated after loading was ∼3 GB.

System Prototype. We run our experiments in an actor database system prototype named REACTDB (described in Chapter 3) with basic implementations of Features 1 to 4. Actors are allocated to thread pools pinned to cores, and all actor state is stored in index structures under optimistic concurrency control (OCC) in Silo [125]. Method invocations on actors provide serializability. We configured the prototype in two modes, namely: (1) sync: all the method invocations across actors are executed synchronously and in the same thread to represent sequential execution; and (2) async: a method invocation on a Store_Section actor is dispatched to the thread pool representing the actor for execution using a queue. We ran our experiments without any durability of transactions. We tuned the thread pool sizes to minimize queuing delays and maximize usage of physical cores.

Methodology. We map each `Cart` actor to the thread pool pinned to each of the eight physical cores in the first socket. We allocate worker threads such that each worker thread generating method invocations on a `Cart` actor is mapped to the hyper-threaded core of the corresponding cart to simulate client affinity. For async, all actors except of type `Store_Section` that are involved in a cart transaction are local to same core as the cart, and accessed synchronously. Invocations to methods of `Store_Section` actors, however, are dispatched for asynchronous execution. Each of the `Store_Section` actors are mapped to thread pools pinned to each of the eight physical cores on the second socket.

A worker runs interactions consisting of (1) `add_items` and on its successful commit (2) `checkout`. We measure the average latency and throughput of the entire interaction using an epoch-based measurement strategy [51]. Each epoch consists of 2 sec, and we report averages and standard deviations of successful interactions over 20 epochs. Workers choose customer IDs from a uniform distribution. The items and store sections in orders are also chosen from a uniform distribution for a configurable number of store sections and items per store section in the order.

2.6.6.2 Leveraging Asynchronicity in SmartMart

To more clearly observe the gains offered by asynchronicity, we first study the effect of increasing both work and asynchronicity in method calls from *a single worker*. We vary the number of store sections from 1 to 8 while keeping the number of items ordered from each section fixed at 4, thus varying the size of the order from 4 to 32. Figure 2.7 shows that the throughput of sync degrades with increasing order size given the sequential execution of the methods. The slope of the curve also decreases with store sections since the increase in the order size is constant, and hence has a smaller impact as the order size grows. By

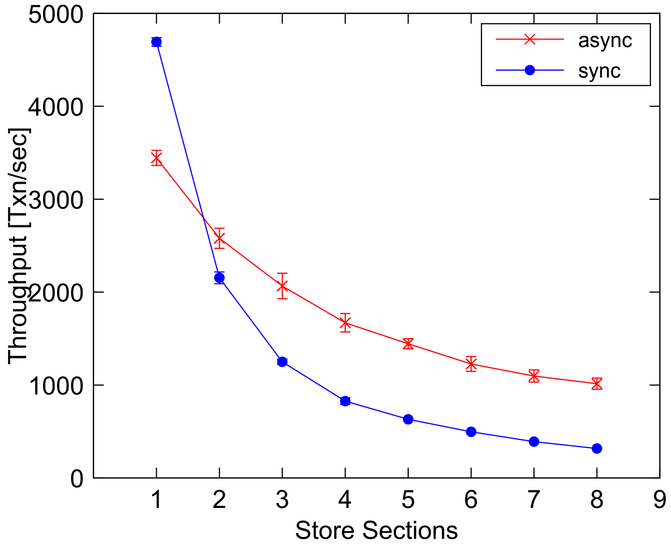


Figure 2.7: SmartMart throughput with varying order size.

contrast, async has lower throughput when the number of store sections is one, but reaches 3.2x higher throughput than sync for 8 store sections. At the beginning, async suffers from lack of sufficient asynchronicity and overhead of dispatch to the `Store_Section` actor as opposed to shared memory accesses in sync. However, as the number of store sections increases, asynchronicity benefits arise since the variable discount computations across store sections during `checkout` and price lookups during `add_items` are overlapped to utilize parallel resources.

2.6.6.3 Effect of Load on Asynchronicity

By gradually increasing the number of concurrent workers, we study the effect of load on the benefits of asynchronicity observed above. We keep the work fixed to an order size of 32, corresponding to an order across 8 store sections and 4 items from each store section, and increase the number of workers, carts and customers in the experiment. Figures 2.8 and 2.9 show the throughput and latency observed. While sync exhibits excellent throughput and latency scalability as we increase the number of workers, the throughput of async scales well until three workers and then degrades before roughly stabilizing. This is because at three workers the `Store_Section` actors are close to full resource utilization (CPU core at 88%), maxing out at four workers and then becoming the bottleneck. The resulting effect of queuing can also be seen in the latency measurements, where the latency increases dramatically after four workers.

Despite the queuing effects, async still outperforms sync because of the amount of physical resources being utilized by it, namely 16 cores with intra-transaction

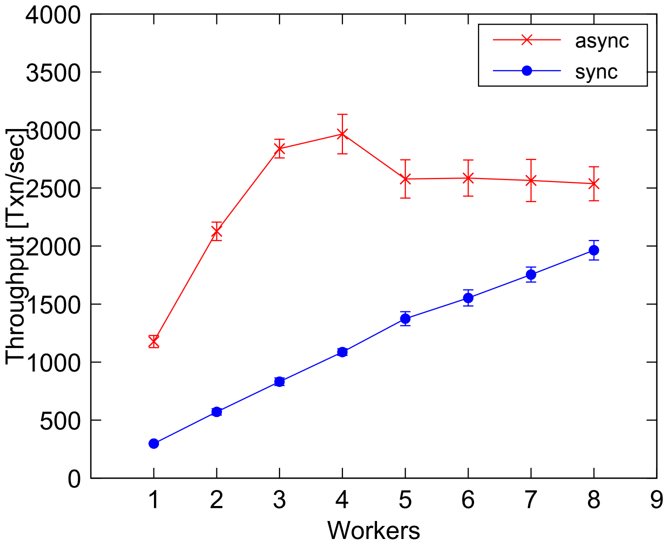


Figure 2.8: SmartMart throughput with fixed order size and varying workers.

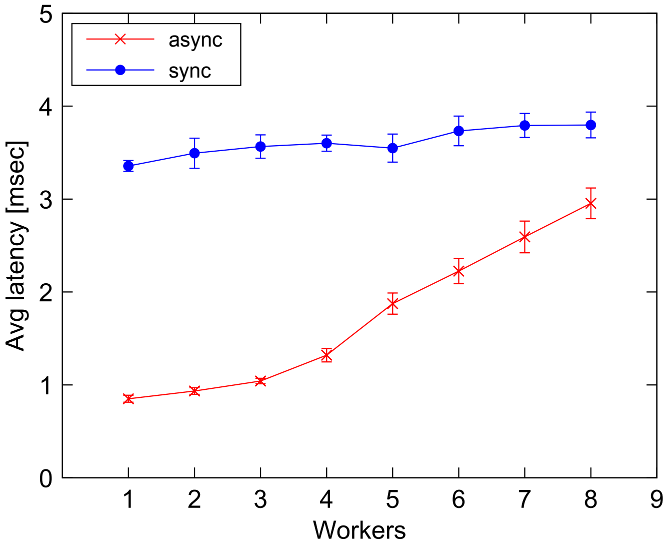


Figure 2.9: SmartMart latency with fixed order size and varying workers.

parallelism as opposed to 8 cores in sequential execution. We did not perform measurements for more than 8 workers, since the hardware does not have enough physical cores to sustain our setting for async. Nevertheless, we would expect a crossover with sync as load increases. In short, asynchronicity can bring both throughput and latency benefits over a traditional synchronous strategy when load in the database is light to normal and transactions exhibit parallelism.

During this experiment, we observed abort rates of ∼5-7% despite the small amount of actual logical contention on items. This happens because the OCC protocol of Silo aborts transactions if the version numbers of nodes scanned change at validation time, caused in our experiments by tree splits due to inserts.

2.6.6.4 SmartMart Implementation Details

In Figures 2.10 and 2.11, and Figure 2.12, we present the pseudocode of the SmartMart application introduced in the main body of the chapter. In the pseudocode, we make use of an additional conversion function `TABLE` to transform a list of values into a relation. The `Customer` and `Group_Manager` actor functionality in Figure 2.10 is straightforward and just interacts with the encapsulated state using declarative queries. In the `Customer` actor, we introduce an annotation to make a relation, `passwd`, and method, `authenticate`, *encrypted* for security.

Figure 2.11 outlines the implementation of the `Store_Section` actor. The `get_price` method returns the minimum price and the price of the requested items from the inventory. The `get_variable_discount_update_inventory` method first computes a relation (`ph`) with the mean and standard deviation of purchase quantities for every item in the list of requested items (`ord_items`) for a statically defined history window size of K. This relation is then joined using an inner join with `inventory` and the relation representing the ordered items (`TABLE(ord_items)`) to get the necessary information required to compute the cumulative price and discounts to be returned for the order. Note that the minimum price has been accounted for in the price and discount computations. Subsequently, for each item in the order, the inventory is updated to reflect the purchase (and replenished if necessary), following which the purchase is recorded in the `purchase_history` relation.

Figure 2.12 shows the implementation of the `Cart` actor. The `add_items` method first constructs a list of item ids by store section from `orders` provided as input. For brevity in the pseudocode, we represented these data structure interactions as function calls, namely: (1) `extract_arrange`, (2) `extract_ids`, and (3) `lookup`. The method then invokes `get_price` method calls on each of the `Store_Section` actors asynchronously storing the futures in a map data structure for later synchronization. Declarative multi-actor queries were not used, because we want to overlap other subsequent computations in the body of the method in addition to the asynchronous `get_price` method calls across store sections.

```
actor Group_Manager {
  state:
    relation discounts (i_id int, fixed_disc float);

  method:

    list<tuple> get_fixed_discounts(list<int> i_ids) {
      return LIST(SELECT * FROM discounts
                  WHERE  i_id IN (TABLE(i_ids)));
    }
};

actor Customer {
  state:
    relation customer_info (cust_name string, c_g_id int);

    relation store_visits (store_id int, time timestamp,
                           amount float,
                           fixed_disc float, var_disc float);

    encrypted relation passwd (enc_passwd string);

  method:

    tuple get_customer_info() {
      SELECT * INTO v_info FROM customer_info;
      return v_info;
    }

    void add_store_visit(int store_id, timestamp time, float amt,
                         float fixed_disc, float var_disc) {
      INSERT INTO store_visits
      VALUES (store_id, time, amt, fixed_disc, var_disc);
    }

    encrypted bool authenticate (string enc_passwd) {
      SELECT * INTO v_passwd FROM passwd;
      return validate_fn(enc_passwd, v_passwd);
    }
};
```

Figure 2.10: Implementation of Group_Manager and Customer actors.

```
actor Store_Section {
  state:
    relation inventory (i_id int, i_price float,
                        i_min_price float,
                        i_quantity int, i_var_disc float);

    relation purchase_history (i_id int, time timestamp,
                               i_quantity int, c_id int);

  method:

    list<tuple> get_price(list<int> i_ids) {
      return LIST(SELECT i_price, i_min_price FROM inventory
                  WHERE i_id IN (TABLE(i_ids)));
     }

    tuple get_variable_discount_update_inventory(
              int c_id, timestamp c_time, list<tuple> ord_items) {
      SELECT SUM((CASE
                   WHEN i_price > i_fixed_disc + i_var_disc
                     THEN i_price - (i_fixed_disc + i_var_disc)
                   ELSE i_min_price) * i_quantity) AS amount,
             SUM(i_fixed_disc * i_quantity) AS fixed_disc,
             SUM((CASE
                    WHEN i_price > i_fixed_disc + i_var_disc
                      THEN i_var_disc
                    ELSE (i_price - i_min_price - i_fixed_disc))
                   * i_quantity) AS var_disc
      INTO v_totals
      FROM
        (SELECT ph.i_id, o.i_quantity,
               (o.i_quantity / (ph.i_avg + c * ph.i_stddev))
                * inv.i_var_disc AS i_var_disc,
                o.i_min_price, o.i_price, o.i_fixed_disc
        FROM (SELECT i_id,
                     AVG(i_quantity)
                         OVER (PARTITION BY i_id
                               ORDER BY time DESC
                               ROWS BETWEEN CURRENT ROW AND K FOLLOWING)
                                 AS i_avg,
                     STDDEV(i_quantity)
                         OVER (PARTITION BY i_id
                               ORDER BY time DESC
                               ROWS BETWEEN CURRENT ROW AND K FOLLOWING)
                                 AS i_stddev
              FROM purchase_history
              WHERE i_id IN (SELECT i_id FROM TABLE(ord_items)) ph
        INNER JOIN TABLE(ord_items) o ON (o.i_id = ph.i_id)
        INNER JOIN inventory inv ON (inv.i_id = ph.i_id));

      foreach o_i IN ord_items {
        UPDATE inventory
        SET i_quantity = CASE
                           WHEN i_quantity > o_i.i_quantity
                             THEN i_quantity - o_i.i_quantity
                           ELSE 10000
        WHERE i_id = o_i.i_id;

        INSERT INTO purchase_history
        VALUES (o_i.i_id, c_time, o_i.i_quantity, c_id);
      }
      return v_totals;
    }
};
```

Figure 2.11: Implementation of Store_Section actor.

```
actor Cart {
  state:
    relation cart_info (c_id int, store_id int, session_id int);

    relation cart_purchases (sec_id int, session_id int, i_id int,
                             i_quantity int, i_fixed_disc float,
                             i_min_price float, i_price float);

  method:
    int add_items(list<order> orders, int o_c_id) {
      // Organize the items ids in orders by store section
      orders_by_store_section = extract_arrange(orders);

      map<int,future> results;
      for (section_order : orders_by_store_section) {
        future res := actor<Store_Section>[section_order.sec_id].
                        get_price(section_order.item_ids);
        results.add(section_order.sec_id, res);
      }

      SELECT c_g_id INTO v_c_g_id
      FROM actor<Customer>.get_customer_info()
      WHERE name = o_c_id;

      // Compute list of all ids of ordered items
      ordered_item_ids :=  extract_ids(orders);
      future disc_res := actor<Group_manager>[v_c_g_id].
                      get_fixed_discounts(ordered_item_ids);

      // Generate session_id and update cart_info
      SELECT session_id + 1 INTO v_session_id FROM cart_info;
      UPDATE cart_info
      SET c_id = o_c_id, session_id = session_id + 1;

      list<tuple> discounts := disc_res.get();
      results.value_list().when_all();

      //Iterate over prices and discounts and store in cart_purchases
      foreach sec_id_res in results {
        foreach i_p in sec_id_res.second.get() {
          fixed_disc := lookup(discounts, i_p.i_id);
          i_quantity := lookup(orders, i_p.i_id);
          INSERT INTO cart_purchases
          VALUES (sec_id_res.first, v_session_id, i_id,
                  i_quantity, fixed_disc, i_p.min_price,
                  i_p.price);
        }
      }
      return v_session_id;
    }

    float checkout(int) {
      ...
    }
};
```

Figure 2.12: Implementation of Cart actor.

After firing price lookups, the customer group is looked up using a declarative multi-actor query on the `Customer` actor. The customer group is then used to invoke `get_fixed_discounts` on the `Group_Manager` actor. Finally, synchronization is used to get the discounts and to wait for all price results to become available from store sections. Using imperative constructs, we iterate over the values of the price results (`results`). For each record in the map data structure, `.first` and `.second` are the handle to the key and the value respectively. For each store section ID (`sec_id_res.first`), we invoke `get()` on the future (`sec_id_res.second`) to get the price values of the items requested from that store section. Note that this call to `get()` returns immediately, since synchronization on all the futures has been done earlier with `when_all()`. We use the price information in conjunction with lookups in our input `orders` and fixed discount values (`discounts`) to then record an entries in `cart_purchases` for use during checkout.

The method `checkout` is implemented in Figure 2.5 and explained in Section 2.6.4. Consequently, the function implementation is omitted from the definition of the `Cart` actor in Figure 2.12 for brevity.

2.7 Research Opportunities

In this section, we outline research avenues in actor database systems organized under topical areas for better perspective.

2.7.1 Theoretical Foundations

With the introduction of actor database systems, interesting challenges arise on how to integrate logical actors, which are compute entities by essence, with the theory of database design and querying [1]. For example, the theory of data normalization has provided a measure of the quality of a database schema, with reduction of data redundancy being a key goal. With the introduction of actors, measures for the quality of a logical actor database (actors + schema) need to be defined. Intuitively, more actors would hint at a more scalable database design, but excessive distribution of application logic may lead to low efficiency. Such trade-offs need to be captured theoretically, and their consequences on the theory of normalization examined. In addition, it is an interesting challenge to formalize the notion of database constraints spanning the schema of relations in multiple actors. In the example of Figure 2.1, in order to ensure that the fixed discount does not exceed an item's minimum price, we would need to specify a constraint spanning `Group_Manager` and `Store_Section` actors.

With the introduction of actors encapsulating state, the classical formalism of transactions also needs to be revisited. The classical transactional model [27] needs extension to formalize transactions across actors and investigate isomorphism of properties of programs across the models. In Chapter 3, we introduce a model for formalizing transactions across actors, called reactors, as an extension to the classical transactional model [27], and prove that serializability of programs is isomorphic across the models. However, formalization of the semantics of detached transactions and of application-defined isolation levels for different child sub-transactions are also open problems that need further exploration.

2.7.2 Conceptual Modeling

The entity-relationship model has become the de facto standard for conceptually modeling the database [39]. The model is connected to a well-understood methodology for translating designs into corresponding database representations. With the introduction of actor databases, similar analytical machinery and associated tools are required to equip application developers with a methodology to model an application using actors.

2.7.3 Programming Model and Query Support

As pointed out in Tenets 2 and 3, actor database systems necessitate the integration of asynchronous programming with declarative querying. It is non-trivial to define proper query semantics and query optimization methods for this new scenario. Another interesting challenge for the programming model is the choice of query capabilities. To support tighter language integration, declarative query capabilities must be exposed by either using native language support or enhancing abstractions supported by the programming language. The space of complete declarative querying versus mixed declarative querying and imperative programming support, as well as the associated impacts on expressibility, productivity, and performance, needs to be explored and evaluated.

2.7.4 System Implementation and Design

Numerous challenges arise on efficiently designing and implementing systems for heterogenous hardware architectures and cloud computing infrastructure while guaranteeing high resource utilization, scalable performance, and ease of programming. Two extreme approaches could be investigated. At one end of the spectrum, one could integrate database features into actor runtimes, starting for example from transaction support [124]. At the other end, one could add actor programming support into classical databases. For the latter, however, classical database components such as logging and recovery would need to be revisited in order to support application-defined durability. At the same time, such cross-cutting low-level mechanisms need to smoothly integrate with potential heterogeneity in actors due to type annotations and with asynchronous execution of methods across actors.

Actor database systems may also open up opportunities to re-architect database systems in new ways. By advocating a design of an application using actors, an actor database system can now introduce a system architecture to virtualize a database across the extremes of shared-everything and shared-nothing at deployment, while keeping the programming model and application programs intact. In Chapter 3, we introduced an architecture to virtualize a database across the extremes of shared-everything and shared-nothing at deployment time, while keeping the programming model and application programs intact. This opens the space of various deployment optimization problems that can be further explored.

2.7.5 Software Engineering and Security

The introduction of actor database systems raises many interesting questions regarding the design of applications across the middle tier and the data tier. The set of design principles that should guide the placement of application functionality needs further exploration. In addition, implications on scalability, resource efficiency, consistency and availability can be explored by segmenting application functionality in different ways between the middle and data tiers. Moreover, the effect of an actor-oriented programming model in the data tier on code quality, i.e., number of bugs or ease of debugging and isolating failures needs to be explored. Furthermore, security models that integrate well with software engineering practices should be investigated. Emerging applications and traditional applications need to be modeled using actor database systems to understand and quantify the benefit of various programming model features.

2.8 Related Work

2.8.1 Actor languages and frameworks

Actors were proposed as a model for concurrent computations centered around a message passing semantics [3]. Actors encapsulate state, provide single-threaded semantics for message handling and hence state manipulation, and support an asynchronous message shipping programming paradigm. Because of these concepts, actor languages and frameworks provide an elegant mechanism to model concurrent and distributed applications [6,32,56,110]. However, managing actor lifecycle, handling faults and ensuring high-performance of actor runtimes in a distributed infrastructure complicates their usage, which has led to the appeal of virtual actors [34] for transactional middleware [24].

Despite advances in actor runtimes such as virtual actors, actors put the burden of state management on the application. Applications need to choose either main memory or using external storage solutions for actor state, depending on durability and fault-tolerance requirements. Applications are also forced to account for and handle the failure and consistency models of the underlying storage systems employed. Lack of all-or-nothing atomicity leads to complications in application code to ensure consistency of application state under failure.

By contrast, actor database systems offer the state management guarantees that classic databases have long provided under the notion of transactions to ensure application developers can focus on writing application logic. By providing well-defined state manipulation semantics in the presence of failures, actor database systems simplify construction of distributed, concurrent and *stateful* applications. In addition, actor runtimes lack a high-level data model along with declarative querying facilities, which actor database systems provide. Actor database systems are envisioned to abstract the data tier as a distributed runtime to increase its programmability and scalability, and not as a replacement of actor runtimes deployed in the middle tier as a soft-caching layer with high-availability and weak-consistency guarantees.

2.8.2 Microservices

Microservices have gained a lot of popularity recently as a software engineering paradigm [69]. Microservices advocate design of software systems as small, modular services that are deployed independently and communicate using a messaging mechanism. Each small, modular service can use an independent software stack. Actor database systems can be viewed as a programming paradigm to design the database tier of a software system using the microservice architecture by functionally decomposing the data tier in modules across actors. By allowing decomposition of the data tier in terms of actors, actor database systems provide a lightweight and resource-efficient primitive when compared with decomposition across multiple database instances as with existing solutions. One or many actor database systems can be used for deployment of the data tier depending on the needs of the application and its design.

2.8.3 Classic Relational Database Management Systems (RDBMS)

RDBMS were designed to support declarative querying of data abstracted using a relational data model [71]. In order to shield the application developer from concurrent execution of application programs and hardware failures, ACID transactions became the de facto standard for RDBMS. At a high level, the programming model of a database is that of a single shared space, where access to data items is achieved using a declarative query language with transactional guarantees. As a performance optimization, stored procedures were introduced to co-locate a sequence of client queries in the database and reduce data transfer costs [108].

This programming model leads to a monolithic design of the data tier causing the following issues: (1) Since any part of the application logic in any stored procedure can access any data stored in relations, it becomes hard to isolate and identify bugs in application logic especially with growing size of data, numbers of relations and stored procedures, and with growing application complexity; (2) Since the entire data and logic are shared in the database, a failure causes unavailability of the entire database system instead of failure of the affected parts only; (3) Since the programming model lacks a notion of an active thread of control and consequently the notion of what constitutes an unit of scalability, it is hard to reason about the scalability of the database without understanding details of database system implementation. In contrast, by providing an actor-oriented primitive for state encapsulation and modularity, actor database systems provide an application-controlled mechanism to functionally decompose the database into modules. This mechanism allows application developers to manage code complexity, isolate bugs, contain failures, and reason about scalability of the database.

2.8.4 Partitioned RDBMS

Modern database systems employ database partitioning to deploy the database over different hardware deployment infrastructures, e.g., multiple machines or multiple cores in the same machine, and co-locate data with processing elements for performance. This technique is applied in database architectures for systems covering the extremes of shared-nothing [80, 121] and shared-everything [50, 76, 98, 125]. Despite the extensive use of data partitioning under the hood in these systems, the programming model introduced by classic RDBMS remains unchanged. As such, partitioned RDBMS also suffer from the same software engineering problems caused by the monolithic design of the data tier, since these systems are invariant in their programming model compared with classic RDBMS. In addition to solving the aforementioned software engineering issues, actor database systems provide a programming model that allows application developers to understand the performance issues with their design. Furthermore, asynchronous messaging between actors allows application developers to explicitly leverage intra-transaction (data and control) parallelism in arbitrary programs, and to reason about the relative performance of different programs depending on the level of parallelism employed.

2.8.5 Object-oriented Database Management Systems (OODBMS)

OODBMS focused on addressing the impedance mismatch existing between RDBMS and programming languages [15, 21]. While persistent programming language runtimes tried to bring database support to popular object-oriented languages such as C++ [16], OODBMS such as O2 proposed an object-oriented data model with an embedded declarative query language [85]. OODBMS proposed object-orientation for modularity, data encapsulation, behavior specification and extensibility. However, objects in OODBMS do not have any notion of a thread of control, i.e, objects are not an active execution entity that can execute logic but rather they are an abstraction to encapsulate data and to define behavior on this data.

On the contrary, actors in an actor database system both encapsulate data and represent an active, concurrently executing entity with a thread of control. This allows for reasoning about locality and scalability in terms of the number of actors and their communication patterns with each other. Actor database systems support asynchronous communication primitives, which allows for specification of parallel programs spanning multiple actors. The latter is not possible with the synchronous method invocation semantics of objects in OODBMS. Actors in actor database systems can support any data model, e.g., the relational data model as advocated in this chapter, while the data model is inflexible in OODBMS.

2.9 Outlook and Conclusion

This chapter has made the case for actor database systems, a new data management approach combining the virtues of actor runtimes and classic databases. Actor database systems comprise logical actors with asynchronous operations as well as transactional features and declarative querying, providing for modularity, parallelism, fault tolerance, and security. We believe that actor database systems open up exciting research possibilities in various aspects of data management ranging from conceptual modeling to system design.

We argue that increasingly the world of interactive data-intensive applications will look more like the scenario depicted in Figure 2.1, where a combination of complex logic and data management is the norm. Instead of having database systems be relegated to persistent state management components in these applications, our call to the database community is for reimagining database programming models and architectures for this new world, and marry actors and database systems into a new abstraction.

Chapter 3

Reactors: A Case For Virtualized, Predictable Actor Databases

> *"The cheapest, fastest, and most reliable components of a computer system are those that aren't there."*
>
> — G. Bell

The requirements for OLTP database systems are becoming ever more demanding. Applications in domains such as finance and computer games increasingly mandate that developers be able to reason about and control transaction latencies in in-memory databases. At the same time, infrastructure engineers in these domains need to experiment with and deploy OLTP database architectures that ensure application scalability and maximize resource utilization in modern machines. In this chapter, we propose a relational actor programming model for in-memory databases as a novel, holistic approach towards fulfilling these challenging requirements. Conceptually, relational actors, or *reactors* for short, are application-defined, isolated logical actors that encapsulate relations and process function calls asynchronously. Reactors ease reasoning about correctness by guaranteeing serializability of application-level function calls. In contrast to classic transactional models, however, reactors allow developers to take advantage of intra-transaction parallelism and state encapsulation in their applications to reduce latency and improve locality. Moreover, reactors enable a new degree of flexibility in database deployment. We present REACTDB, a system design exposing reactors that allows for flexible virtualization of database architecture between the extremes of shared-nothing and shared-everything without changes to application code. Our experiments with REACTDB illustrate latency control, multi-core scalability, and low overhead in OLTP benchmarks.

3.1 Introduction

Three trends are transforming the landscape of OLTP systems. First, a host of latency-sensitive OLTP applications has emerged in areas as diverse as computer games, high-performance trading, and web applications [33, 120, 132]. This trend brings about challenging performance requirements, including mechanisms to allow developers to reason about transaction latencies and scalability of their applications with large data and request volumes [114, 126]. Second, database systems are moving towards solid state, in particular in-memory storage [82], and hardware systems are integrating increasingly more cores in a single machine. This trend brings about new requirements for database architecture, such as processing efficiency in multi-core machines and careful design of concurrency control strategies [125, 134]. Third, there is a need to operate databases out of virtualized infrastructures with high resource efficiency [25, 81, 94]. This trend leads to the requirement that virtualization abstractions for databases impose low overhead and allow for flexible deployments without causing changes to application programs.

Recent research in OLTP databases has shown that addressing all of these requirements is a hard problem. On the one hand, shared-nothing databases, such as H-Store [121] or HyPer [80], fail to provide appropriately for multi-core efficiency. This is due to the impact of overheads in mapping partitions to cores and of synchronous communication in distributed transactions across partitions. Consequently, these systems are very sensitive to how data is partitioned, with impacts on both transaction latencies and throughput. On the other hand, shared-everything databases have a hard time achieving multi-core scalability. To do so, these systems either internally partition their data structures, e.g., DORA [98], or rely heavily on affinity of memory accesses to cores in transactions submitted to the system, e.g., Silo [125]. Thus, deployment decisions can deeply affect efficiency and scalability in these systems and are difficult to get right across application classes.

As a consequence, both developers and infrastructure engineers in demanding OLTP domains have a hard time controlling the performance of their transactional databases. Despite advances in profiling tools to identify causes of latency variance in database systems [73], today developers lack clear abstractions to reason about transaction latencies at a high level in their applications. In addition, the variety of modern in-memory database engines, including numerous specialized designs ranging internally from shared-nothing to shared-everything [99, 106, 133], challenges the ability of infrastructure engineers to flexibly experiment with and adapt database architecture without affecting application code.

Actor programming models provide desirable primitives for concurrent and distributed programming [3, 14, 70], which of late have evoked a strong interest in the database community [26]. To holistically meet the challenging requirements imposed on OLTP systems, we propose a new actor programming model in relational databases called *Relational Actors* (or *reactors* for short). Reactors are special types of actors that model logical computational entities encapsulating state abstracted as relations. For example, reactors can represent application-level scaling units such as accounts in a banking application or warehouses in a retail management application. Within a reactor, developers can take advantage of classic database programming features such as declarative querying over the encapsulated relations. To operate on state logically distributed across reactors, however, developers employ explicit asynchronous function calls. The latter allows developers of latency-sensitive OLTP applications to write their programs so as to minimize cross-reactor accesses or overlap communication with computation. Still, a transaction across multiple reactors provides serializability guarantees as in traditional databases, thus relieving developers from reasoning about complex concurrency issues. With reactors, developers can reason about latency vs. scalability trade-offs in their transactional applications as well as about relative performance of different programs. Reactors allow application-level modeling between the extremes of a relational database (single reactor encapsulating all relations) and key-value stores (each reactor encapsulating a key-value pair).

To address the challenges of architectural flexibility and high resource efficiency in multi-core machines, we design an in-memory database system that exposes reactors as a programming model. This system, called REACTDB (RElational ACTor DataBase), decomposes and virtualizes the notions of sharing of compute and memory in database architecture. First, we introduce a containerization scheme to abstract shared-memory regions in a machine. Second, within a container, compute resources abstracted as transaction executors can be deployed to either share or own reactors. The combination of these two notions allows infrastructure engineers to experiment with deployments capturing a range of database architecture patterns in a manner that is as simple as changing a configuration file. At the same time, no changes are required to application code using reactors.

Example: Digital Currency Exchange. We abstract an application coded using a set of reactors as a *reactor database*. Consider a simplified digital currency exchange application, in which users may buy or sell currency through their credit card providers. Figure 3.1 contrasts how such an application would be written with a classic transactional database and a reactor database in parts (a) and (b), respectively. The exchange wishes to bound its exposure to settlement risk, namely cancelled credit card payments, by a maximum limit stored in relation `settlement_risk`. In part (a), it does so in the procedure `auth_pay` by only allowing additions to the `orders` relation if the total current exposure is below the limit. In part (b), we see the same logic expressed with reactors. The exchange and each of the providers are modeled as relational actors with private state (relations) that can execute certain procedures. The exchange reactor can execute `auth_pay` and encapsulates in its state information about providers and the settlement limit. Provider reactors store in their states fragments of the relation `orders` with the payments for each provider, and can execute procedures `calc_risk` and `add_entry`. In `auth_pay`, the exchange reactor invokes asynchronous calls to provider reactors, making explicit the available intra-transaction parallelism. Since the exchange strives for the lowest latency possible, this program formulation improves transaction response time with respect to part (a) in a way that is clearly explainable to a developer pursuing application performance optimization. In addition, it becomes explicit in transaction programs that code is conceptually moved close to the data it touches, allowing developers to control for locality. At the same time, ACID properties are guaranteed for `auth_pay`, providing the same level of safety as in the classic transactional model, but with higher performance control. As evidenced by the collapse of the Flexcoin bitcoin exchange [116], we argue that the right trade-off for new OLTP applications is to provide abstractions that allow developers to organize application logic for performance *without* compromising consistency in concurrent executions. We further elaborate on this example and our model in Section 3.2.

How are reactors different from database partitioning?

In contrast to database partitioning, which is a data-oriented optimization, reactors represent a *compute-oriented abstraction*. Reactors can be used to model horizontal partitioning of tuples, but also vertical partitioning of relations, or any arbitrary grouping of relation fragments. However, reactors can also model functional decomposition of an application. For example in a banking application, one reactor can capture functionality of debits and credits over accounts, while another the functionality of risk calculations to authorize debits and credits, which is not feasible to model with database partitioning alone. In addition, reactors provide a primitive to model affinity and parallelism in arbitrary application logic. For example, the risk calculations can done by different prediction models in parallel in the database by running these calculations in different reactors. By contrast, detecting this parallelism in a classic stored procedure over a single logical global schema would require complex control-flow analysis, and may not be possible at all.

Contributions. In summary, our work makes the following contributions:

1. We present a novel logical abstraction for relational databases called *reactors*. This abstraction is grounded on transactional semantics offering serializability and an asynchronous programming model allowing for *reasoning at a high level about latency and application scalability* (Section 3.2).

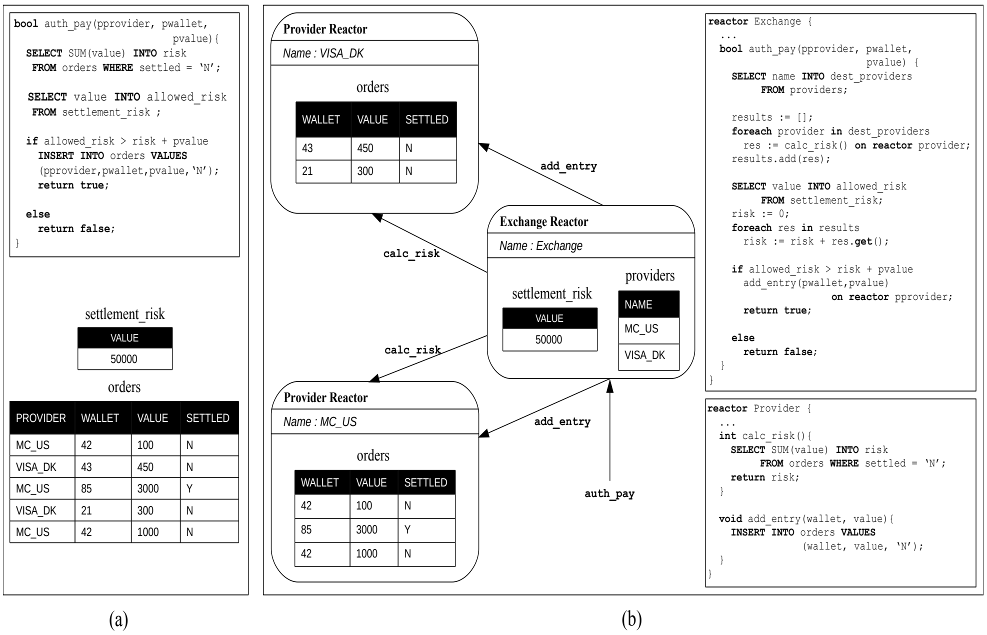
2. We discuss the design of ReactDB, an in-memory database system for OLTP exposing reactors. ReactDB enables *configuration of database architecture at deployment time* without changes to application code (Section 3.3).

3. In experiments with classic OLTP benchmarks, reactors provide latency control at the microsecond scale for varied program formulations. In addition, for given program formulations, database architecture can be configured to allow for scalable execution in a multi-core machine (Section 3.4).

3.2 Programming Model

3.2.1 Reactor Concepts

In contrast to classic transactional or actor models, reactors bring together all of the following concepts:



```
bool auth_pay(pprovider, pwallet,
                              pvalue){
  SELECT SUM(value) INTO risk
   FROM orders WHERE settled = 'N';

  SELECT value INTO allowed_risk
   FROM settlement_risk ;

  if allowed_risk > risk + pvalue
    INSERT INTO orders VALUES
    (pprovider,pwallet,pvalue,'N');
    return true;

  else
    return false;
}
```

settlement_risk

VALUE
50000

orders

PROVIDER	WALLET	VALUE	SETTLED
MC_US	42	100	N
VISA_DK	43	450	N
MC_US	85	3000	Y
VISA_DK	21	300	N
MC_US	42	1000	N

(a)

orders

WALLET	VALUE	SETTLED
43	450	N
21	300	N

settlement_risk

VALUE
50000

providers

NAME
MC_US
VISA_DK

orders

WALLET	VALUE	SETTLED
42	100	N
85	3000	Y
42	1000	N

```
reactor Exchange {
  ...
  bool auth_pay(pprovider, pwallet,
                              pvalue) {
    SELECT name INTO dest_providers
         FROM providers;

    results := [];
    foreach provider in dest_providers
      res := calc_risk() on reactor provider;
    results.add(res);

    SELECT value INTO allowed_risk
         FROM settlement_risk;
    risk := 0;
    foreach res in results
      risk := risk + res.get();

    if allowed_risk > risk + pvalue
      add_entry(pwallet,pvalue)
                    on reactor pprovider;
      return true;

    else
      return false;
  }
}
```

```
reactor Provider {
  ...
  int calc_risk(){
    SELECT SUM(value) INTO risk
         FROM orders WHERE settled = 'N';
    return risk;
  }

  void add_entry(wallet, value){
    INSERT INTO orders VALUES
                  (wallet, value, 'N');
  }
}
```

(b)

Figure 3.1: A simplified currency exchange application in: (a) the classic transactional model, and (b) the reactor model.

1. A reactor is an application-defined logical actor that encapsulates state abstracted using relations.

2. Declarative queries are supported only on a single reactor. Communication across reactors is achieved by asynchronous function calls. A computation (function) across reactors consists of a sequence of intra-reactor statements and/or nested cross-reactor function calls.

3. Computations (functions) across reactors provide transactional guarantees.

4. Reactors provide an abstract computational cost model to allow for high-level reasoning about transaction latencies.

3.2.2 Programming with Reactors

3.2.2.1 Application-Defined Relational Actors

A *reactor* is a type of actor specialized for the management of state abstracted by the relational model. The pseudocode in Figure 3.2 conceptualizes the capabilities of a reactor. As a regular actor [3], a reactor encapsulates a state, which can be accessed by computations invoked on the reactor. However, unlike in a regular actor, in which communication is typically achieved by non-blocking send and blocking

```
Reactor : Actor {
    RelationalState rs;

    Future execute(compute_fn, args) {
        return new Future(compute_fn(args,rs));
    }
}
```

Figure 3.2: Conceptual view of reactors as an actor type.

receive primitives, the only form of communication with a reactor is through *asynchronous function calls* returning promises [89]. Moreover, the code of such functions is similar to that of database stored procedures, which can intermix declarative queries over relations with other program logic and function calls. These asynchronous function calls are abstracted in Figure 3.2 by the `execute` function, which takes as an argument a function to be computed on the reactor's state along with appropriate arguments, and returns a promise representing the result of the computation. In the remainder of this chapter, we refer to such a result as a *future*, and use the terms *function* and *procedure* on a reactor interchangeably.

A *reactor database* is a collection of reactors. In order to instantiate a reactor database, we need to declare the names of the reactors constituting the reactor database and a schema creation function to define the relations encapsulated by each reactor type. A reactor is a purely logical computational entity and can be accessed by the name declared when initializing the reactor database. The application developer has full control over the schema layouts for the reactors. The developer cannot create or destroy reactors; declaring the name of the reactor ensures that it is available for the lifetime of the application, bound by the failure model of the reactor database. Note that in contrast to objects in an object-oriented database [35, 85], reactors are active computational entities, i.e., a reactor is a combination of a logical thread of control and an encapsulated relational state accessible *exclusively* by that logical thread. While objects encapsulate complex types, reactors encapsulate whole relational schemas; declarative querying happens only within, not across reactors, and communication across reactors is explicit through asynchronous function calls.

In the example of Figure 3.1(b), the state of a provider reactor consists of a horizontal fragment of the `orders` relation, but with the `provider` column removed (since the provider is the same for all tuples in such a reactor). The state of the exchange reactor consists of relations `settlement_risk` and `providers`. This illustrates that different reactors may contain either the same or different schemas. The application logic in procedure `auth_pay` first looks up the providers to calculate the risks for. The procedure then performs a nested asynchronous invocation of the procedure `calc_risk` on each of the provider reactors.

It is not necessary to know in advance all the providers and their names to model the reactor database. It is sufficient to know: (1) the types of the reactors expected, namely exchange and provider reactors; (2) the schema of each reactor type; and (3) the name mapping to address provider reactors. As such, adding new providers does not necessitate rewriting the application logic.

3.2.2.2 Asynchronous Function Calls

To invoke a procedure on a reactor, we must explicitly use the declared name of the reactor where the computation must be executed. The procedure logic can access the relational state on the reactor where it is invoked through declarative queries. If the procedure needs to access the state of another reactor, then it must invoke another procedure on the target reactor. This is necessary because the states of different reactors are disjoint. Since the result of a procedure is represented by a future, the calling code can choose to wait for the result of the future, invoke procedures on other reactors, or execute further application logic. This flexibility allows application developers to expose parallelism within the procedure.

In Figure 3.1(b), the syntax `procedure_name(args) on reactor reactor_name` specifies an asynchronous procedure call routed to the reactor with a given name. In the logic of `auth_pay`, the execution of `calc_risk` is overlapped on each of the provider reactors and the futures returned by the calls are stored in the `results` list. The exchange reactor then looks up the allowed risk value to further overlap the execution of this logic before synchronizing on the results of the futures. The exchange reactor sums up the total risk by accessing the value of each future by invoking `get()` on the future object. If the total risk is within the allowed risk, then the exchange reactor performs another nested asynchronous procedure call to `add_entry` on the provider reactor name given as a parameter to `auth_pay`. This call results in adding an order at the appropriate target provider reactor. Asynchronous procedure calls allow the application logic to leverage available parallelism in a computation spanning reactors.

Does the reactor programming model necessitate manual optimization? We posit that reactors can act as a bridging model between a classic database abstraction and a key-value API. Reactors provide the possibility for developers to navigate the extremes between a single-reactor with full SQL support and a radical decomposition of individual tuples as reactors. We envision that the typical application modeling will be a hybrid between these two extremes balancing reuse of query optimization machinery with low-level performance control. The popularity of NoSQL databases points to the need and willingness among application developers to obtain higher performance control and scalability for their applications even at the cost of sacrificing traditional database features such as query optimization and transaction support.

3.2.2.3 Reactor Consistency using Transactional Semantics

To guarantee consistency of the state encapsulated by a reactor database, the semantics of procedure invocations on reactors is transactional. We differentiate between top-level and nested asynchronous procedure calls. Top-level calls are executed by clients on a reactor and are termed interchangeably *transactions* or *root transactions*. Transactions respect the classic ACID properties: atomicity, consistency, isolation, and durability [27]. We denote a concrete execution i of a transaction by T_i.

Nested asynchronous procedure calls are executed by a reactor on another reactor. Since these calls must always occur within the overall context of a root transaction, they are called *sub-transactions*. We denote a concrete execution j of a sub-transaction of transaction T_i on a reactor k by $ST^k_{i,j}$. Sub-transactions allow programmers to structure their computations for performance, allowing for concurrent computation on (logically) distributed state among reactors. Sub-transactions are *not* used, however, to allow for partial commitment. Any condition leading to an abort in a sub-transaction leads to the abort of the corresponding root transaction. This approach towards the semantics of nested calls is exactly the reverse of what is adopted in classic systems such as Argus [88], reflecting our focus on leveraging the high degree of *physical parallelism* in modern commodity hardware for transactional processing as opposed to managing faults in settings with a high degree of physical distribution (e.g., geo-distribution) as in previous work. A transaction or sub-transaction completes only when all its nested sub-transactions complete. This frees the client logic from explicitly synchronizing on the result of a sub-transaction invocation if it does not need the result of the sub-transaction.

For example in Figure 3.1(b), the `auth_pay` procedure does not wait on the result of the `add_entry` procedure call, since the programming model guarantees that the transaction corresponding to `auth_pay` only completes when all its sub-transactions complete.

Any program in the classic transactional model can be trivially remodeled in the reactor programming model by specifying a single reactor. For example, we could model a single exchange reactor with the schema and application logic shown in Figure 3.1(a). However, the benefits of our programming model are only fully achieved when developers of latency-sensitive OLTP applications remodel their logic as done in Figure 3.1(b). In particular, in the reformulated logic, intra-transaction parallelism is exposed. Furthermore, the trade-off between scalability on the number of provider reactors and latency of executing the logic of `auth_pay` becomes explicit.

Is there a development methodology to architect an application using reactors? An interesting avenue for future research is to explore an analytical machinery allowing for modeling and comparing the quality of reactor database designs, similar to the entity relationship model and decomposition of universal relations by functional dependencies in classic relational databases. Such an analytical machinery could provide application developers with a methodology to model their applications as reactors. Although answering this question is beyond the scope of this chapter, we envision that the following steps could be taken by developers. As a first step, the developer starts from a single reactor with the whole relational schema and all application functions. Unlike in a traditional relational database, however, as the developer observes undesirable latency or throughput scalability, now s/he has the possibility to break down complex application functions across multiple reactors. Throughout this process of performance measurement and functionality decomposition, the developer does not need to worry about correctness issues due to concurrency, which are handled by the transactional semantics in the reactor programming model. This process exposes latency costs as cross-reactor communication that can be overlapped, and enables scaling on the number of reactors. This exercise can also point developers to inherent scalability limitations of the application itself.

3.2.2.4 Intra-Transaction Safety

Introducing asynchronicity in a transactional abstraction is not trivial. Since asynchronicity exposes intra-transaction parallelism, race conditions could arise when sub-transactions that conflict on a data item are invoked asynchronously on the same reactor. Moreover, such invocations would violate the illusion that a reactor is a computational entity with a single logical thread of control. To avoid these issues, we must enforce that at most one execution context is active for a given reactor and root transaction at any time.

First, we enforce that whenever a reactor running a procedure directly executes a nested procedure invocation on itself, the nested invocation is executed *synchronously*. This policy corresponds to inlining the sub-transaction call, resulting in future results being immediately available. To deal with nested asynchronous invocations, we define the *active set* of a reactor k as the set of sub-transactions, regardless of corresponding root transaction, that are currently being executed on reactor k, i.e., have been invoked, but have not completed. Thus, the runtime system must conservatively disallow execution of a sub-transaction $ST_{i,j}^{k}$ when:

$$\exists ST_{i,j'}^{k} \in \text{active\_set}(k) \ \wedge \ j' \neq j$$

This dynamic safety condition prohibits programs with cyclic execution structures across reactors and programs in which different paths of asynchronous calls lead to concurrent sub-transactions on the same reactor. By introducing this safety condition, the run-time conservatively assumes that conflicts may arise in asynchronous accesses to the same reactor state within a transaction, and thus aborts any transaction with such dangerous structures.

By leveraging this dynamic safety condition, we envision that appropriate testing of transaction logic at development time will be sufficient to root out most, if not all, dangerous structures from the code of latency-sensitive OLTP applications. However, formalizing static program checks to aid in detection of dangerous call structures among reactors is an interesting direction for future work.

3.2.3 Conflict-Serializability of Transactions

To formalize the correctness of concurrent executions of transactions in reactors, we show equivalence of serializable histories in the reactor model to serializable histories in the classic transactional model. We restrict ourselves exclusively to the notion of conflict-serializability. Technically, our formalization is similar to reasoning on nonlayered object transaction models [129].

3.2.3.1 Background

We first review the formalism introduced by Bernstein et al. [27, page 27] for the classic transactional model and introduce relevant notation. In this model, the database consists of a collection of named data items, and transactions encapsulate a sequence of operations. A transaction T_i is formalized as a partial ordering of operations with an ordering relation $<_i$ and comprises a set of operations. Operations include reads and writes, along with either a commit or an abort. A read from a data item x is denoted $r_i[x]$, a write to x denoted $w_i[x]$, while a commit is denoted c_i and an abort a_i. The ordering relation $<_i$ orders *conflicts*. Two operations conflict iff at least one of them is a write and both of them reference the same named item. We assume that a transaction does not contain multiple operations of the same type to the same named data item as in [27, page 27] without any impact on the results.

3.2.3.2 Reactor Model

Without loss of generality, we assume reactor names to be drawn from the set of natural numbers. Recall that we denote a sub-transaction operation in transaction T_i on reactor k by $ST^k_{i,j}$, where j identifies the sub-transaction within the transaction T_i. $r^k_{i,j}[x]$ denotes a read from data item x, and $w^k_{i,j}[x]$ denotes a write to data item x in $ST^k_{i,j}$. Note that data items in different reactors are disjoint. Using this notation, we can define a sub-transaction and a transaction in the *reactor model* as follows.

Definition 1. A sub-transaction $ST^k_{i,j}$ is a partial order with ordering relation $<_{i,j}$ where,

1. $ST^k_{i,j} \subseteq \{\, r^k_{i,j}[x],\ w^k_{i,j}[x],\ ST^{k'}_{i,j'} \mid j \neq j'$, x is a data item in k $\}$;

2. Let

$$\begin{aligned}\text{basic\_ops}(r^k_{i,j}[x]) &= r^k_{i,j}[x]\\ \text{basic\_ops}(w^k_{i,j}[x]) &= w^k_{i,j}[x]\\ \text{basic\_ops}(ST^k_{i,j}) &= \{\text{basic\_ops(o)} \mid o \in ST^k_{i,j}\},\end{aligned}$$

if $o_1 \in ST^k_{i,j} \wedge o_2 \in ST^k_{i,j}$
$\wedge\, r^{k'}_{i,j'}[x] \in \text{basic\_ops}(o_1)$
$\wedge\, w^{k'}_{i,j''}[x] \in \text{basic\_ops}(o_2)$,
then either $o_1 <_{i,j} o_2$ or $o_2 <_{i,j} o_1$.

Note that the ordering relation $<_{i,j}$ of a sub-transaction establishes order according to conflicts in leaf-level basic operations, potentially nested in sub-transactions.

Definition 2. A transaction T_i is a partial order with ordering relation $<_i$ where,

1. $T_i \subseteq \{\, ST^k_{i,j} \,\} \cup \{a_i, c_i\}$;

2. $a_i \in T_i \iff c_i \notin T_i$;

3. if t is $c_i\ or\ a_i$ (whichever is in T_i), for any further operation
   $p \in T_i, p <_i t$;

4. if $o_1 \in T_i \wedge o_2 \in T_i \wedge o_1, o_2 \notin \{a_i, c_i\} \wedge$
   $r^{k'}_{i,j'}[x] \in \text{basic\_ops}(o_1) \wedge w^{k'}_{i,j''}[x] \in \text{basic\_ops}(o_2)$,
   then either $o_1 <_i o_2$ or $o_2 <_i o_1$.

Formally, a transaction comprises exclusively sub-transactions, and the relation $<_i$ orders sub-transactions according to conflicts in their nested basic operations. In the reactor model, two sub-transactions *conflict* iff the basic operations of at least one of them contain a write and the basic operations of both of them reference the same named item in the same reactor. Under this extended notion of a conflict, the definition of history, serial history, equivalence of histories and serializable history in the reactor model are the same as their definitions in the classic transactional model [27], but with sub-transactions replacing basic operations. With these definitions in place, we now show that serializability of transactions in the reactor model is equivalent to serializability of an appropriately defined projection into the classic transactional model.

Definition 3. The projection of a basic operation o from the reactor model to the classic transactional model, denoted by $P(o)$, is defined as:

1. $P(r^k_{i,j}[x]) = r_i[k \circ x]$

2. $P(w^k_{i,j}[x]) = w_i[k \circ x]$

3. $P(c_i) = c_i$

4. $P(a_i) = a_i$

where $\circ$ denotes concatenation.

The definition provides a name mapping from the partitioned address space of reactors to a single address space, which is done by concatenating the reactor identifier with name for a data item.

Definition 4. The projection of a sub-transaction $ST_{i,j}^{k}$ from the reactor model to the classic transactional model, denoted by $P_S(ST_{i,j}^{k})$ is a partial order with ordering relation $<_S^{i,j}$:

1. $P_S(ST_{i,j}^{k}) \subseteq \{P(o) \mid o \in \text{basic\_ops}(ST_{i,j}^{k})\}$;
2. if $o_1 \in ST_{i,j}^{k} \wedge o_2 \in ST_{i,j}^{k} \wedge\ o_1 <_{i,j} o_2\ \wedge\ o_1, o_2$ are reads or writes, then $P(o_1) <_S^{i,j} P(o_2)$;
3. if $ST_{i,j'}^{k'} \in ST_{i,j}^{k}$, then $<_S^{i,j}$ is extended by $<_S^{i,j'}$;
4. if $o_1 \in ST_{i,j}^{k} \wedge o_1$ is a read or a write $\wedge\, ST_{i,j'}^{k'} \in ST_{i,j}^{k} \wedge\ o_1 <_{i,j} ST_{i,j'}^{k'}$, then $P(o_1) <_S^{i,j} P_S(ST_{i,j'}^{k'})$;
5. if $ST_{i,j'}^{k'} \in ST_{i,j}^{k} \wedge o_2 \in ST_{i,j}^{k} \wedge o_2$ is a read or a write $\wedge\, ST_{i,j'}^{k'} <_{i,j} o_2$, then $P_S(ST_{i,j'}^{k'}) <_S^{i,j} P(o_2)$;
6. if $ST_{i,j'}^{k'} \in ST_{i,j}^{k} \wedge ST_{i,j''}^{k''} \in ST_{i,j}^{k} \wedge ST_{i,j'}^{k'} <_{i,j} ST_{i,j''}^{k''}$, then $P_S(ST_{i,j'}^{k'}) <_S^{i,j} P_S(ST_{i,j''}^{k''})$.

Definition 5. The projection of a transaction T_i from the reactor model to the classic transactional model, denoted by $P_T(T_i)$ is a partial order with ordering relation $<_T^{i}$:

1. $P_T(T_i) \subseteq (\bigcup_{ST_{i,j}^{k} \in T_i} P_S(ST_{i,j}^{k})) \cup \{P(o) \mid\ o \in T_i \wedge o \text{ is a commit or abort}\}$;
2. $<_T^{i}$ is extended by $\bigcup_{ST_{i,j}^{k} \in T_i} <_S^{i,j}$;
3. if $ST_{i,j}^{k} \in T_i \wedge ST_{i,j'}^{k'} \in T_i \wedge ST_{i,j}^{k} <_i ST_{i,j'}^{k'} \wedge o_1 \in P_S(ST_{i,j}^{k}) \wedge o_2 \in P_S(ST_{i,j'}^{k'})$, then $o_1 <_T^{i} o_2$;
4. if t is $c_i\ or\ a_i$ (whichever is in $P_T(T_i)$), for any further operation $p \in P_T(T_i), p <_T^{i} t.$

The definitions unroll all sub-transactions in the reactor model into read and write operations in the classic transactional model while maintaining ordering constraints.

Definition 6. The projection of a history H over a set of transactions T = $\{T_1, T_2, ..., T_n\}$, in the reactor model to the classic transactional model, denoted by $P(H)$ is a partial order with ordering relation $<_{P_H}$ over a set of transactions $T' = \{P_T(T_1), P_T(T_2), ..., P_T(T_n)\}$ iff:

1. $P_{P_H}(H) = \bigcup_{i=1}^{n} P_T(T_i)$;
2. $<_{P_H}$ is extended by $\bigcup_{i=1}^{n} <_T^i$;
3. if $o_1 \in P_S(ST_{i,j}^k) \wedge o_2 \in P_S(ST_{i',j'}^{k'}) \wedge ST_{i,j}^k \in T_i \wedge ST_{i',j'}^{k'} \in T_{i'} \wedge ST_{i,j}^k <_H ST_{i',j'}^{k'}$, then $o_1 <_{P_H} o_2$ as long as o_1 and o_2 conflict.

Theorem 3.2.1. *A history H is serializable in the reactor model iff its projection $H' = P(H)$ in the classic transactional model is serializable.*

Proof. Let us assume H is serializable and H' is not serializable. From the serializability theorem, since H is serializable, the serializability graph of H $(SG(H))$ is acyclic; since the projected history H' is not serializable, the serializability graph $SG(H')$ must be cyclic. Therefore, there must exist a cycle $T_i' \rightarrow \ldots \rightarrow T_j' \rightarrow \ldots \rightarrow T_i'$. Since the graph is built on operations of the classic transactional model, then there must be conflicting operations $o_i' <_{P_H} \ldots <_{P_H} o_j' <_{P_H} \ldots <_{P_H} o_i'$. By condition (3) of Definition 6, there must exist sub-transactions $ST_{i,l}^k \in T_i$ and $ST_{j,l'}^{k'} \in T_j$ such that $ST_{i,l}^k <_H \ldots <_H ST_{j,l'}^{k'} <_H \ldots <_H ST_{i,l}^k$. As a result, $SG(H)$ must be cyclic, and we arrive at a contradiction. To show the reverse direction, it is simple to follow a similar argument, but starting with a cyclic graph in SG(H) and showing that SG(H') must be cyclic as well in contradiction. □

Theorem 3.2.1 implies that we can, with appropriate care, employ an existing scheduler for the classic transactional model to implement a scheduler for the reactor model. This is done in Section 3.3 by reusing the optimistic concurrency control (OCC) scheduler of Silo [125] together with the two-phase commit (2PC) protocol [27].

3.2.4 Computational Cost Model

In this section, we introduce a cost model to support developers of latency-sensitive OLTP applications in controlling the latency of a transaction program expressed using reactors. Clearly, latency depends heavily on program structure. For example, certain programs can overlap asynchronous invocations of functions in other reactors with processing logic and/or synchronous function invocations; other programs may do so only conditionally, or have data dependencies between different asynchronous function calls. For concreteness, we focus on a subset of programs where in every sub-transaction, all asynchronous invocations happen simultaneously at one given program point only, but our cost analysis can be extended to other program structures as well.

Consider a sub-transaction $ST_{i,j}^k$. We call $\text{sync}_{seq}(ST_{i,j}^k)$ its sequence of children sub-transactions and $P_{seq}(ST_{i,j}^k)$ its processing logic executed synchronously and not overlapped with asynchronous sub-transactions. Any sub-transaction invocation incurs communication costs. We term $C_s(k, k')$ the cost to send a sub-transaction call from reactor k to reactor k', and $C_r(k', k)$ the cost to receive a result

$$
\begin{aligned}
L(ST^{k}_{i,j}) = P_{seq}(ST^{k}_{i,j}) \quad &+ \sum_{ST^{k'}_{i,j'} \in \, \mathrm{sync}_{seq}(ST^{k}_{i,j})} L(ST^{k'}_{i,j'}) \\
+ \sum_{k' \in \, \mathrm{dest}\big(\mathrm{sync}_{seq}(ST^{k}_{i,j})\big)} & \big(C_s(k,k') + C_r(k',k)\big) \\
+ \; max\Bigg(max_{ST^{k'}_{i,j'} \in \, \mathrm{async}(ST^{k}_{i,j})} & \Bigg(L(ST^{k'}_{i,j'}) + C_r(k',k) \\
& + \sum_{k'' \in \, \mathrm{dest}\big(\mathrm{prefix}(\mathrm{async}(ST^{k}_{i,j})\,,\, ST^{k'}_{i,j'})\big)} C_s(k,k'')\Bigg), \\
P_{ovp}(ST^{k}_{i,j}) \quad & + \sum_{ST^{k'}_{i,j'} \in \, \mathrm{sync}_{ovp}(ST^{k}_{i,j})} L(ST^{k'}_{i,j'}) \quad + \\
\sum_{k' \in \, \mathrm{dest}\big(\mathrm{sync}_{ovp}(ST^{k}_{i,j})\big)} & \big(C_s(k,k') + C_r(k',k)\big)\Bigg)
\end{aligned}
$$

Figure 3.3: Modeling latency cost of a procedure call in the reactor model.

from k' at k. The sequence of children sub-transactions of $ST^{k}_{i,j}$ executed asynchronously are denoted async($ST^{k}_{i,j}$). The synchronous children sub-transactions and processing logic overlapped with the asynchronous sub-transactions are represented by sync$_{ovp}$($ST^{k}_{i,j}$) and $P_{ovp}(ST^{k}_{i,j})$, respectively. If S represents a sequence of sub-transactions, prefix($S, ST^{k}_{i,j}$) denotes the sequence of sub-transactions in S upto $ST^{k}_{i,j}$ and including it. Moreover, we say that dest($ST_1, \ldots, ST_n$) represents the sequence of reactors that sub-transactions $ST_1, \ldots, ST_n$ execute on.

Now, the latency cost of $ST^{k}_{i,j}$ is modeled by the formula in Figure 3.3. The formula represents the latency cost if enough parallelism is available in the underlying physical implementation to overlap all asynchronous sub-transactions. The same formula can be applied recursively to compute the latency cost for sub-transactions of arbitrary depth. Since a root transaction is a special case of a sub-transaction, i.e., a sub-transaction without a parent, the same formula applies, modulo any overheads incurred for commitment.

Reasoning about cost components as in Figure 3.3, developers can re-architect their programs to improve the latency of their sub-transactions by: (1) increasing asynchronicity of children sub-transactions, (2) overlapping execution of application logic by introducing sub-transactions, and (3) reducing the processing cost of the application logic. In addition, without worrying about absolute values of cost parameters, but considering that communication events have in general larger cost than local processing, the developer can reason about relative latency costs for her program in a way similar to algorithmic complexity measures. In particular, when communication cost dwarfs local processing cost, the reasoning is similar to what is done under the I/O model [2].

Finally, developers can reason about scalability by considering how data, computation and communication are spread among reactors. In particular, if developers architect their applications such that increasing amounts of data and computation are distributed among increasing numbers of reactors while at the same time keeping the number of cross-reactor calls roughly constant per transaction, then adequate transactional scalability should be expected.

As explained in the discussion on manual optimization in Section 3.2.2.2, reactors allow developers to navigate a space of choices between fully automatic optimization and manual performance-oriented design. The existing approach of fully automatic optimization does not provide a clean methodology to deal with performance problems when optimization fails. With reactors, however, developers can reason at a high-level on latency cost and employ reactor design and asynchronous function calls to better control performance, allowing specification of application-level information otherwise unavailable to the database system.

3.3 System Architecture

3.3.1 Overview

In this section, we discuss the architecture of REACTDB, an in-memory database system that exposes the reactor programming model. The design of REACTDB aims at providing control over the mapping of reactors to physical computational resources and memory regions under concurrency control. The system implementation currently targets a single multi-core machine for deployment; however, REACTDB's architecture is designed to allow for deployments in a cluster of machines, which we leave for future work.

As shown in Figure 3.4, REACTDB's architecture is organized as a collection of *containers*. A container abstracts a (portion of a) machine with its own storage (main memory) and associated mechanisms for transactional consistency. Each container is isolated and does not share the data stored in it with other containers. Containers are associated with computational resources (cores) disjoint from other containers, abstracted by *transaction executors*. A transaction executor consists of a thread pool and a request queue, and is responsible for executing requests, namely asynchronous procedure calls. Each transaction executor is pinned to a core. Single-container transactions are managed by the concurrency control mechanism within the container, while a *transaction coordinator* runs a commitment protocol for transactions spanning multiple containers. Transactional durability is currently disabled in our implementation, but could be achieved by a combination of techniques such as fast log-based recovery [137] and distributed

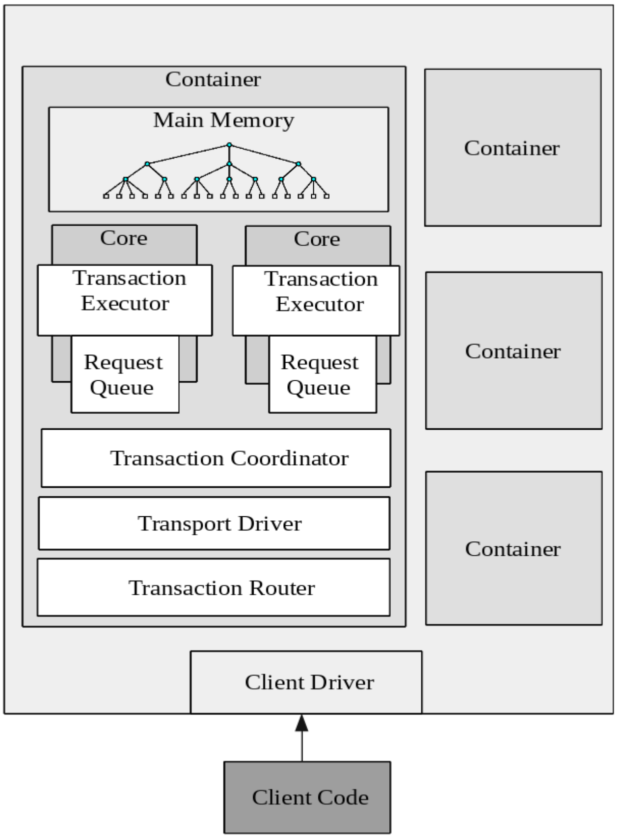


Figure 3.4: REACTDB's Architecture.

checkpoints [55]. Alternatively, an approach such as FaRM's could be employed to minimize any impact of durability mechanisms on latency [52].

Each container stores a two-level mapping between a reactor and a transaction executor. On the first level, a reactor is mapped to one and only one container. Together with appropriate container deployment, this constraint ensures that asymmetrically large communication costs are only introduced between, but not within, reactors, in line with our computational cost model. On the second level, a reactor can be mapped to one or more transaction executors in a container. *Transaction routers* decide the transaction executor that should run a sub-transaction according to a given policy, e.g., round-robin or affinity-based.

Transport drivers handle communication across containers. REACTDB has a driver component that is used by client code to send transactions into the system for processing. REACTDB accepts pre-compiled stored procedures written in the reactor programming model in C++ against a record manager interface. An instance of a pre-compiled stored procedure and its input forms a transaction.

3.3.2 Concurrency Control

3.3.2.1 Single Container Transactions

Every transaction or sub-transaction written in the reactor programming model specifies the reactor where it must be executed. If the destination reactor of a child sub-transaction is hosted in the same container as the parent sub-transaction, the child sub-transaction is executed synchronously within the same transaction executor to minimize the communication overhead of migrating across transaction executors. If all the sub-transactions in the execution context of a root transaction are executed within one container, then the native concurrency control mechanism of the container is used to guarantee serializability. As a consequence of Theorem 3.2.1, REACTDB can reuse an existing concurrency control mechanism, and we chose Silo's high-performance OCC implementation [125].

3.3.2.2 Multi-Container Transactions

When a sub-transaction is invoked on a reactor mapped to a container different than the current container, the call is routed by the transport driver to the destination container and then by the transaction router to the request queue of a transaction executor. Once the sub-transaction is queued, the calling code gets a future back representing this computation. If the calling sub-transaction code does not synchronize on the future, then once the caller completes, REACTDB enforces synchronization on the futures of all child sub-transactions.

Two-Phase Commit. By the above synchronization policy, a root transaction can finish when all the sub-transactions created and invoked in its context finish, recursively. The transaction executor then invokes the transaction coordinator to initiate a commitment protocol across the containers that have been touched by the transaction, either directly or by any of its deeply nested sub-transactions. The transaction coordinator in turn performs a 2PC protocol. The first phase of the protocol runs a validation of Silo's OCC protocol on all the involved containers. If the validations are successful, the locks on the write-set of the transaction are acquired and released only when the write phase ends. If any of the validations fail, the transaction is aborted.

Cooperative Multitasking. To minimize the effect of stalls due to synchronization, each transaction executor maintains a thread pool to process (sub-)transactions. The threads use cooperative multitasking to minimize context switching overheads. A thread blocks if it tries to access the result of a sub-transaction invoked on a different container and the result is not yet available. In such a situation, it notifies another thread to take over processing of the request queue and goes back to the thread pool when the (sub-)transaction being executed by it is completed.

3.3.3 Deployments

The decomposition of the notions of compute and shared memory by transaction executors and containers allows infrastructure engineers to flexibly deploy REACTDB in a number of database architectures. In the remainder of the chapter, we restrict ourselves to three main deployment strategies:

(S1) shared-everything-without-affinity: This strategy employs a single container in which each transaction executor can handle transactions on behalf of any reactor. REACTDB is configured with a round-robin router to load balance transactions among executors. All sub-transactions are executed within the same transaction executor to avoid any migration of control overhead. This strategy adheres to the architecture of most shared-everything databases [71].

(S2) shared-everything-with-affinity: This strategy is similar to shared-everything-without-affinity in that it employs a single container, but with the difference that an affinity-based router ensures that root transactions for a given reactor are processed by the same transaction executor. In sub-transaction calls, even if to different reactors, no migration of control happens, and the sub-transaction is executed by the same transaction executor of the root transaction. This deployment strategy closely adheres to the setup employed in the evaluation of Silo [125].

(S3) shared-nothing: This strategy employs as many containers as transaction executors, and a given reactor is mapped to exactly one transaction executor. While this strategy aims at maximizing program-to-data affinity, sub-transaction calls to different reactors may imply migration of control overheads to other transaction executors. In our experiments (Section 3.4), we further decompose this configuration into shared-nothing-sync and shared-nothing-async, depending on how sub-transactions are invoked within application programs. In the former option, sub-transactions are invoked synchronously by calling `get` on the sub-transaction's future immediately after invocation. In the latter option, the call to `get` is delayed as much as possible for maximal overlapping of application logic with sub-transaction calls. From an architecture perspective, both of these setups represent a shared-nothing deployment with differing application programs exercising different synchronization options. The deployment strategy shared-nothing-sync models the setup of shared-nothing databases such as H-Store [121] and HyPer [80], albeit with a different concurrency control protocol. The shared-nothing-async strategy represents a deployment that allows REACTDB to leverage intra-transaction parallelism as provided by the reactor programming model.

Other flexible deployments, similar to [104], are possible as well. To change database architecture, only configuration files need to be edited and the system bootstrapped. Since applications operate only on reactors, the mapping of reactors to containers and transaction executors is immaterial for the application logic.

3.4 Evaluation

In this section, we evaluate the effectiveness of REACTDB and the reactor programming model. The experiments broadly aim at validating the following hypotheses: **(H1)** The reactor programming model allows for reasoning about latency in alternative formulations of application programs (Section 3.4.2.1). **(H2)** The computational cost model of reactors can be efficiently realized by REACTDB (Sections 3.4.2.3 and 3.4.2.2). **(H3)** REACTDB exhibits close to linear transactional scale-up in a standard benchmark when deployed across multiple cores (Sections 3.4.3.1 and 3.4.3.2). **(H4)** REACTDB allows for configuration of database architecture, without any changes to application code, so as to exploit asynchronicity in transactions depending on the level of load imposed on the database (Section 3.4.3.3 and 3.4.3.4).

3.4.1 Experimental Setup

3.4.1.1 Hardware

For our latency measurements in Section 3.4.2, we employ a machine with one four-core, 3.6 GHz Intel Xeon E3-1276 processor with hyperthreading, leading to a total of eight hardware threads. Each physical core has a private 32 KB L1 cache and a private 256 KB L2 cache. All the cores share a last-level L3 cache of 8 MB. The machine has 32 GB of RAM and runs 64-bit Linux 4.1.2. A machine with high clock frequency and uniform memory access was chosen for these experiments to challenge our system's ability to reflect low-level latency asymmetries in modern hardware as captured by our programming model.

For our measurements in Section 3.4.3, we use a machine with two sockets, each with 8-core 2.1 GHz AMD Opteron 6274 processors including two physical threads per core, leading to a total of 32 hardware threads. Each physical thread has a private 16 KB L1 data cache. Each physical core has a private 64 KB L1 instruction cache and a 2 MB L2 cache. Each of the two sockets has a 6 MB L3 cache. The machine has 125 GB of RAM in total, with half the memory attached to each of the two sockets, and runs 64-bit Linux 4.1.15. The higher number of hardware threads and accentuated cache coherence and cross-core synchronization effects allow us to demonstrate the effect of virtualization of database architecture in scalability and cross-reactor transaction experiments.

3.4.1.2 Methodology

An epoch-based measurement approach similar to Oltpbench is used [51]. Average latency or throughput is calculated across 50 epochs and the standard deviation is plotted in error bars. All measurements include the time to generate transaction inputs.

3.4.1.3 Workloads and Deployments

For the experiments of Section 3.4.2, we implement an extended version of the Smallbank benchmark mix [9]. Smallbank simulates a banking application where customers access their savings and checking accounts. Oltpbench first extended this benchmark with a transfer transaction, which is implemented by a credit to a destination account and a debit from a source account [64]. We extend the benchmark further with a multi-transfer transaction. Multi-transfer simulates a group-based transfer, i.e., multiple transfers from the same source to multiple destinations. Thus, by varying the number of destination accounts for multi-transfer and controlling the deployment of REACTDB, we can vary both the amount of processing in the transaction as well as the amount of cross-reactor accesses that the transaction makes.

Each customer is modeled as a reactor. We configure REACTDB with 7 database containers, each hosting a single transaction executor for a total of 7 transaction executors mapped to 7 hardware threads. The deployment plan of REACTDB is configured so that each container holds a range of 1000 reactors. A single worker thread is employed to eliminate interference effects and allow us to measure latency overheads of single transactions. The worker thread generating transaction inputs and invocations is allocated in a separate worker container and pinned to the same physical core hosting the container responsible for the first range, but in a separate hardware thread. In order to keep our measurements comparable, the multi-transfer transaction input generator always chooses a source customer account from this first container.

The experiments of Section 3.4.3 use the classic TPC-C benchmark [123]. We closely follow the implementation of the benchmark from Oltpbench [64], which makes usual simplifications, e.g., regarding think times. In our port of TPC-C, we model each warehouse as a reactor. We configure the number of transaction executors to be equal to the scale factor for the experiment. The number of client worker threads generating transaction calls is also equal to the scale factor, and these workers are configured to reside in a separate worker container. Each client worker thread generates load for only one warehouse (reactor), thus modeling client affinity to a warehouse.

To showcase REACTDB's ability to configure database architecture at deployment time, we experiment with the deployments described in Section 3.3.3.

3.4.1.4 Application Programs

We evaluate different application program formulations for the multi-transfer transaction added to Smallbank, exercising the asynchronous programming features of reactors (see Section 3.4.2.4). Similar to Figure 3.1(b), multi-transfer invokes multiple sub-transactions. In contrast to the figure, in some program variants, we force synchronous execution by immediately calling `get` on the future returned. The first formulation, fully-sync, invokes multiple transfer sub-transactions from

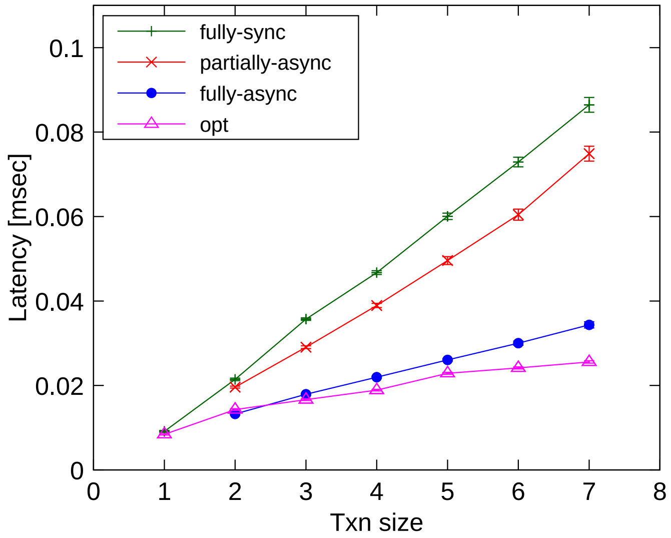


Figure 3.5: Latency vs. size and different user program formulations.

the same source synchronously. Each transfer sub-transaction in turn invokes a synchronous credit sub-transaction on the destination account and a synchronous debit sub-transaction on the source account. The partially-async formulation behaves similarly; however, each transfer sub-transaction invokes an asynchronous credit on the destination account and a synchronous debit on the source account, overlapping half of the writes in the processing logic while still executing communication proportional to the transaction size sequentially. The fully-async formulation does not invoke transfer sub-transactions, but rather explicitly invokes asynchronous credit sub-transactions on the destination accounts and multiple synchronous debit sub-transaction on the source account. Thus, not only are roughly half of the writes overlapped, but also a substantial part of the communication across reactors. The final formulation, opt, is similar to the fully-async transaction, but performs a single synchronous debit to the source account for the full amount instead of multiple synchronous debits. Consequently, processing depth is further reduced and should roughly equal two writes, while communication should be largely overlapped.

In addition, we implement all transactions of TPC-C in our programming model. Unless otherwise stated, we always overlap calls between reactors as much as possible in transaction logic by invoking sub-transactions across reactors asynchronously.

3.4.2 Latency Control

3.4.2.1 Latency vs. Program Formulations

In this section, we show an experiment in which we vary the size of a multi-transfer transaction by increasing the number of destination accounts. Each destination is chosen on a different container out of the seven in our shared-nothing deployment (see Section 3.4.2.2 for alternative choices). The latency for the different application program formulations is outlined in Figure 3.5. The observed curves match the trends predicted by the cost equation of Figure 3.3. First, as we increase transaction size, the processing and communication costs of a multi-transfer increase linearly across all formulations. Second, the highest latencies overall are for fully-sync, and latencies become lower as more asynchronicity is introduced in the formulations by overlapping sub-transaction execution. Third, there is a substantial gap between partially-async and fully-async, due to asymmetric costs between receiving procedure results and sending procedure invocations to other reactors. The latter manifests because of thread switching costs across cores in the receive code path, as opposed to atomic operations in the send code path. In opt, latency is further reduced when compared to fully-async by cutting in almost half the processing costs, which have a smaller impact than communication across cores. It is interesting to note that these optimizations can be done on the μsec scale. The programming model allows a developer to reduce the latency of a transaction from 86 μsec to 25 μsec by simple program reformulations without compromising consistency.

3.4.2.2 Latency Control across Physical Configurations

In this section, we complement the results in Section 3.4.2 by evaluating the latency impact of different physical mappings of reactors in REACTDB. In particular, we provide additional experiments showing that observed latencies for the multi-transfer transaction can be reliably explained by whether the affected reactors are co-located or not on physical components employed in a given configuration.

3.4.2.2.1 Local vs. Remote Calls In this section, we show how the configuration of physical distribution can affect the latency of transactions. The cost equation of Figure 3.3 models communication costs among reactors (namely C_s and C_r), which are higher when reactors are mapped to containers over distinct physical processing elements. We term calls among such physically distributed reactors *remote calls*. By contrast, calls between reactors mapped to the same container are termed *local calls*.

To highlight cost differences in remote calls, we consider the fully-sync and opt multi-transfer formulations. We evaluate two extremes: either destination accounts span all containers (-remote) or are on the same container as the source account (-local). Figure 3.6 shows that the cost of fully-sync-remote rises sharply

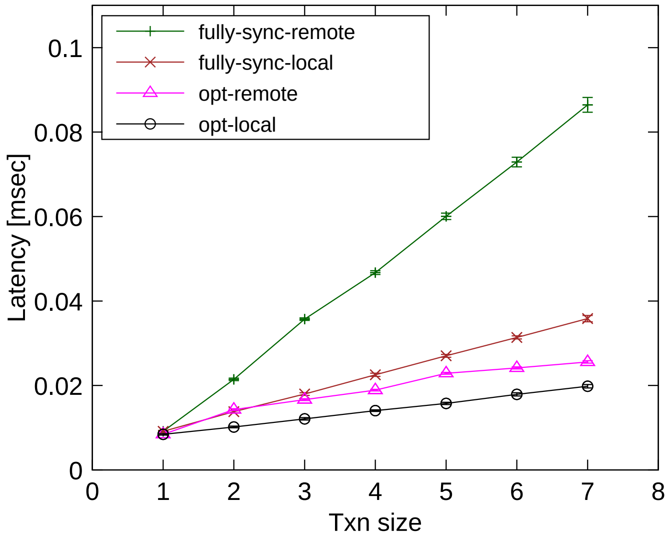


Figure 3.6: Latency vs. size and different target reactors spanned.

because of increase in both processing and communication costs compared to fully-sync-local, which only sees an increase in processing cost. There is a comparatively small difference between opt-local and opt-remote, since the processing of remote credit sub-transactions is overlapped with the local debit sub-transaction, and thus part of the fourth component summed in Figure 3.3. The extra overhead in opt-remote comes from larger, even if partially overlapped, communication and synchronization overheads to invoke the sub-transactions on the remote transaction executors and receive results.

3.4.2.2.2 Varying Degree of Physical Distribution In order to better understand the growth in communication costs due to remote calls, we conduct another experiment where we fix the multi-transfer transaction size to seven destination accounts, and then control these accounts so as to span a variable number of containers. Recall that in the deployment for this experiment, each of the seven containers has exactly one transaction executor pinned to a hardware thread. We use the fully-sync formulation of multi-transfer, so we expect to see higher latencies as a larger number of the credits to the seven destination accounts are handled by remote transaction executors.

We experiment with three variations for selecting destination accounts for our multi-transfer transaction as we vary the number k of transaction executors spanned from one to seven. The first variant, round-robin remote, performs $7-k+1$ local debit calls by choosing accounts mapped to the first container, and $k-1$ remote calls by choosing accounts round-robin among the remaining containers. The second variant, round-robin all, performs $\lceil 7/k \rceil$ local calls and $\lfloor 7/k \rfloor$ remote

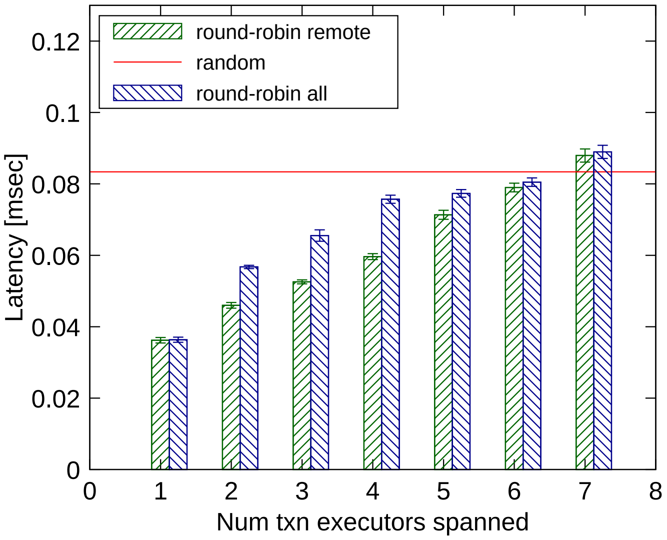


Figure 3.7: Latency for transactions of fixed size varies predictably with distribution of target reactors.

calls. Finally, we measure an expected value for latency by selecting destination accounts with a uniform distribution, termed random.

Figure 3.7 shows the resulting latencies. We observe a smooth growth in the latency for round-robin remote, since we increase the number of remote calls exactly by one as we increase the number of transaction executors spanned. The behavior for round-robin all differs in an interesting way. For two transaction executors spanned, round-robin all performs three remote calls and four local calls. While round-robin all performs four remote calls and three local calls for three transaction executors, the method performs five remote calls and two local calls for both five and six transaction executors spanned. These effects are clearly reflected in the measured latencies. For random, the expected number of remote calls is between six and seven, which is again tightly confirmed by the latency of the multi-transfer transaction in Figure 3.7.

3.4.2.3 Cost Model Breakdown

In this section, we break down our measurements of transaction latencies by the cost components in Figure 3.3, and further validate that our cost model is realizable in ReactDB. We focus on the fully-sync and opt multi-transfer formulations described above, and vary the size of the multi-transfer transaction by changing the number of destination accounts similarly to Figure 3.5. For each variant, we profiled the execution time of the programs in ReactDB into the components of the cost model of Figure 3.3. In addition, we used the profiling information from fully-sync for a transaction size of one to calibrate the parameters of the

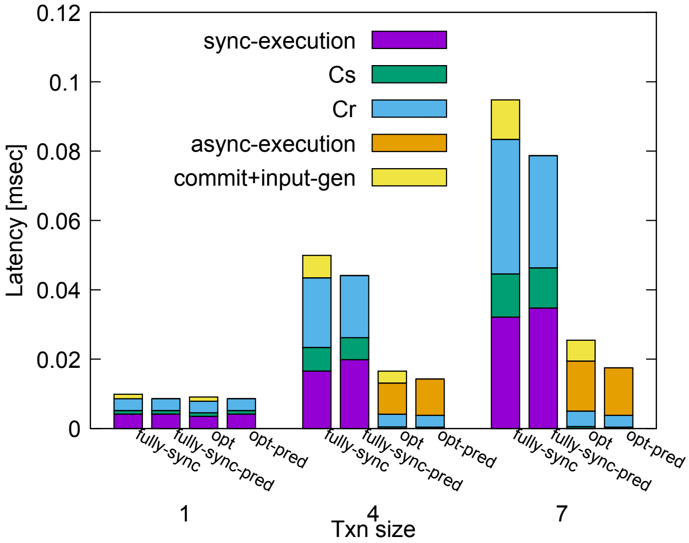


Figure 3.8: Latency breakdown into cost model components.

cost model for prediction, including processing and communication costs. From the parameter values for the single-transfer fully-sync run, we employed the cost equation of Figure 3.3 to predict the execution costs for other transaction sizes and for both the fully-sync and opt program formulations. The predicted values are labeled fully-sync-pred and opt-pred.

We break down transaction latencies into the following components: (a) *sync-execution*: the cost of processing the logic in the transaction and in synchronous sub-transactions, corresponding to the first two components of the cost equation in Figure 3.3; (b) C_s and C_r: the forward and backward costs of communication between reactors in the third component of the cost equation; (c) *async-execution*: the cumulative execution cost of all asynchronous sub-transactions overlapped with any synchronous sub-transactions and processing logic, corresponding to the fourth component of the cost equation; (d) *commit + input-gen*: the cost of the commit protocol, including OCC and 2PC, along with the time to generate the inputs for the transaction. The latter cost component is not shown in Figure 3.3 since it only applies to root transactions and not to any sub-transaction in the reactor programming model. As such, we would expect the bulk of the difference between the predicted and observed performance to be explainable by this cost component.

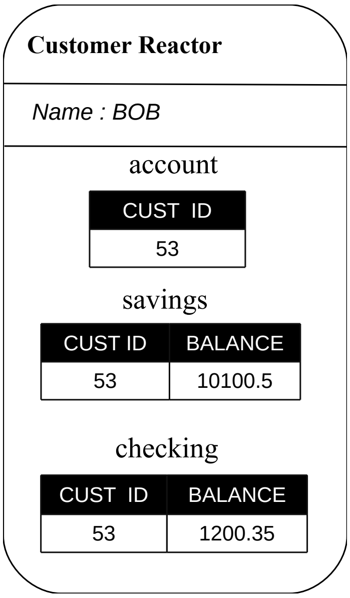
Figure 3.8 shows that the predicted breakdown for the cost components closely matches the latencies profiled for actual executions in ReactDB, even at such a fine granularity. The slight difference in the overlap of different bars is within the variance of the observed vs. the calibration measurement, and expected especially since calibration measures parameters within the 5μsec range. For a transaction size of one, we can see that opt has the same performance behavior as fully-sync.

This effect arises because the destination transaction executor for the credit in the transfer is the same as the source transaction executor for the debit, resulting in a synchronous execution of the credit and debit sub-transactions similar to fully-sync. As we increase the transaction size, the number of transaction executors spanned by the transaction increases, and the execution costs of fully-sync grow because of increasing costs in *sync-execution*, C_s and C_r. As remarked in Section 3.4.2.1, the cost asymmetry between C_s and C_r arises due to thread switching costs across cores in the receive code path compared to atomic operations in the send code path. For opt, we do not observe any *sync-execution* costs, since all credit sub-transactions are overlapped with each other and with the single debit on the source reactor. The growth in the *async-execution* cost of opt with increasing transaction size is caused by the rising communication cost for the sequence of credit sub-transactions, i.e, the last asynchronous credit sub-transaction incurs a cumulative cost of communication of all asynchronous sub-transactions before it in the sequence.

In summary, we observe that reactors and asynchronous function calls, the two primary constructs of our programming model, can be used to explain effects on the latency of transactions at the microsecond scale. In particular, the number of calls to distributed reactors and the degree of call overlap strongly influence transaction latency. The latencies of transactions can be reliably profiled in our system REACTDB into the components of the cost model of Figure 3.3 even in a challenging scenario where the cost of the processing logic in the benchmark is extremely small and comparable to the cost of communication across reactors. This evidence indicates that it is possible for developers to observe and explain the latency behavior of programs with reactors and to reformulate their programs for better performance.

3.4.2.4 Smallbank Implementation Details

In this section, we provide implementation details of the application programs in the Smallbank benchmark, which were used in the experiments in Sections 3.4.2.1, 3.4.2.3 and 3.4.2.2. In this benchmark, each customer was modeled as a reactor. Figure 3.9 outlines the encapsulated relations on each `Customer` reactor, namely (1) `account`, which maps the customer name to a customer ID, (2) `savings` and (3) `checking`, which represent the saving and checking account of the customer, respectively. For strict compliance with the benchmark specifications, we have maintained the customer ID field in the savings and checking relations despite it being a relation holding a single tuple for a customer reactor. We have also performed the lookup on the `account` relation for customer ID followed by its use on the `saving` and `checking` relation to maintain the logic and query footprint of the benchmark specification. Figure 3.10 shows the various methods exposed by the customer reactor to represent the various formulations of the multi-transfer application logic.



CUST ID
53

CUST ID	BALANCE
53	10100.5

CUST ID	BALANCE
53	1200.35

Figure 3.9: Example Customer Reactor in Smallbank.

The `transfer` transaction uses an enviroment variable during compile time (`env_seq_transfer`) that can be enabled or disabled to execute the `multi_transfer_sync` method in fully-sync or partially-sync mode.[1] In the benchmark, `multi_transfer_sync`, `multi_transfer_fully_async`, and `multi_transfer_opt` were invoked on the source customer reactor from which the amount must be transferred to the destination customer reactors. The explicit synchronization in `multi_transfer_sync` is done for safety, though not required when the `src_cust_name` customer reactor executes the method. This is because in this case the nested `transact_saving` sub-transaction on the `src_cust_name` reactor is executed synchronously producing the same effect. However, explicitly specifying the synchronization there improves code clarity. For the same reason, explicit synchronization on the `transact_saving` sub-transaction is done in the `multi_transfer_fully_async` method as well.

3.4.3 Virtualization of Database Architecture

3.4.3.1 Transactional Scale-Up

In this section, we evaluate the scalability of REACTDB across multiple cores for the three database architecture deployments described in Section 3.3.3. Figures 3.11 and 3.12 show the average transaction throughputs and latencies of running the TPC-C transaction mix as we increase the number of warehouses (reactors). We observe that the shared-everything-without-affinity deployment exhibits the worst

[1] This also helps in minimizing code duplication.

```
reactor Customer {
  …
  void transact_saving(amt) {
    SELECT cust_id INTO v_cust_id FROM account;
    SELECT balance INTO v_bal FROM savings WHERE cust_id = v_cust_id;

    if v_bal + amt < 0
      abort;

    UPDATE savings SET balance = balance + amt WHERE cust_id = v_cust_id;
  }

  void transfer(src_cust_name, dst_cust_name, amt) {
    if amt <= 0
      abort;

    res := transact_saving(dst_cust_name, amt) on reactor dst_cust_name;

    if env_seq_transfer
      res.get();

    transact_saving(src_cust_name, -amt) on reactor src_cust_name;
  }

   void multi_transfer_sync(src_cust_name, dst_cust_names, amt){
    foreach dst_cust_name in dst_cust_names
      res := transfer(src_cust_name, dst_cust_cust_name, amt)
                         on reactor src_cust_name;
      res.get();
  }

  void multi_transfer_fully_async(src_cust_name, dst_cust_names, amt) {
    if amt <= 0
      abort;

    foreach dst_cust_name in dst_cust_names
      transact_saving(dst_cust_name, amt) on reactor dst_cust_name;

    foreach val in dst_cust_names
      res := transact_saving(src_cust_name, -amt) on reactor src_cust_name;
      res.get();
  }

  void multi_transfer_opt(src_cust_name, dst_cust_names, amt) {
    if amt <= 0
      abort;

    foreach dst_cust_name in dst_cust_names
      transact_saving(dst_cust_name, amt) on reactor dst_cust_name;

    num_dsts := dst_cust_names.size();
    transact_saving(src_cust_name, -(amt*num_dsts)) on reactor src_cust_name;
  }
}
```

Figure 3.10: Implementation of Smallbank multi-transfer transactions.

throughput and latency scalability among the deployments selected. This effect is a consequence of shared-everything-without-affinity's poor ability to exploit memory access affinities within each transaction executor, given round-robin routing of transactions. On the other hand, shared-everything-with-affinity and shared-

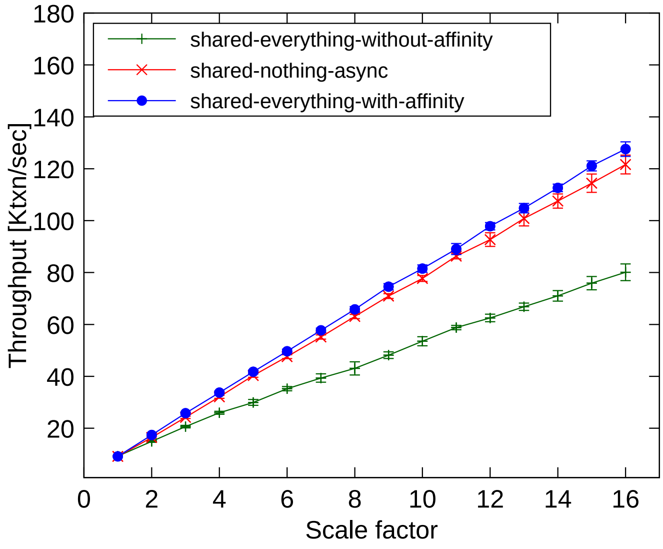


Figure 3.11: TPC-C throughput with varying deployments.

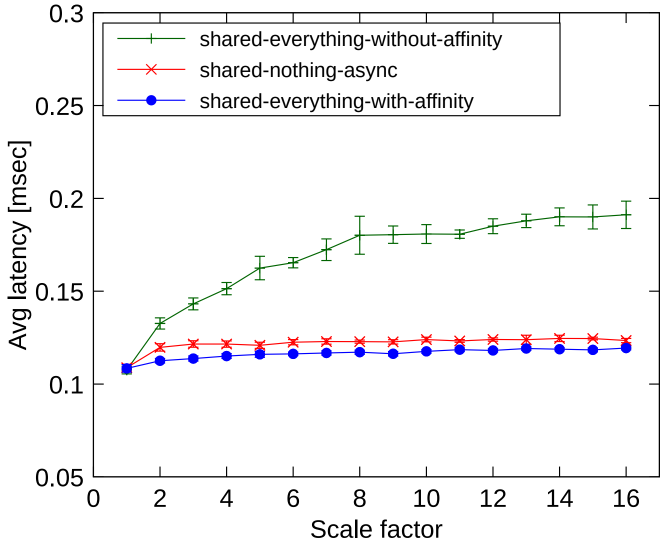


Figure 3.12: TPC-C latency with varying deployments.

nothing-async both take advantage of access affinities and behave similarly. We see that shared-everything-with-affinity is slightly superior to shared-nothing-async. The difference lies in the relative costs in these deployments of sub-transaction invocations vs. direct memory access of data for remote warehouses. For a scale factor of one, there are no cross-reactor transactions, and the performance of the two deployments is identical. From a scale factor of two onwards, the probabilities of cross-reactor transactions range between 0% to 10% (there is a 1% chance of items in new-order being remote and a 15% chance for customer lookups in payment). In shared-nothing-async, a sub-transaction call is routed to its corresponding transaction executor, incurring context switching and communication overheads. By contrast, since shared-everything-with-affinity executes the sub-transaction in the same transaction executor, the remote call costs are traded off for the relatively smaller costs of cache pressure. We also ran the experiment with all the transaction classes in the TPC-C mix invoking sub-transactions synchronously in the shared-nothing deployment (shared-nothing-sync configuration described in Section 3.3.3). However, the throughput and latency of this configuration was close (within the variance bars) to the shared-nothing-async configuration because of the low percentage of cross-container calls in the default TPC-C mix. We hence omit the curve from Figures 3.11 and 3.12 for brevity.

In short, the results indicate that ReactDB can be flexibly configured with different database architectures to achieve adequate transactional scalability for a given workload. In the case of TPC-C, high affinity of data accesses in transactions to physical processing elements (cores) is fundamental to performance, and observed under all configurations explored but shared-everything-without-affinity, which models the architecture of most shared-everything databases. Further exploration of the issue of affinity and of containerization overheads of ReactDB is presented in Section 3.4.3.2.

3.4.3.2 Affinity and Overhead in ReactDB

In this section, we complement the results in Section 3.4.3.1 by delving into the effect of affinity on the shared-everything-without-affinity deployment and characterizing the containerization overheads of ReactDB.

3.4.3.2.1 Effect of Affinity To further drill down into the issue of affinity of reactors to transaction executors, we ran an experiment in which we vary the number of transaction executors deployed in shared-everything-without-affinity, but keep the scale factor of TPC-C at one with a single client worker. The results are shown in Figure 3.13. In such a setup, for k transaction executors deployed, the load balancing router ensures the n-th request is sent to transaction executor $n \mod k$. Thus, the different transactions from the workers are being spread around the transaction executors, which destroys the locality in the transaction executors and accentuates the cache coherence and cross-core communication costs. We found that with two transaction executors throughput drops to 86%

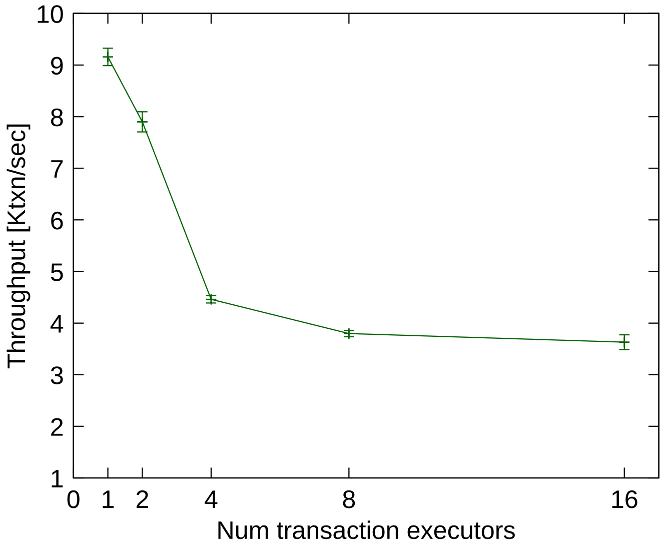


Figure 3.13: Effect of affinity on TPC-C throughput for shared-everything-without-affinity.

compared to one transaction executor and progressively degrades to 40% for 16 transaction executors. For comparison, the corresponding per-core throughput for shared-everything-with-affinity at scale factor 16 in Figure 3.11 is 87% of the per-core throughput at scale factor one. This result highlights the importance of maintaining affinity of transaction execution for high performance, especially in a NUMA machine with accentuated cache coherence and cross-core communication costs.

3.4.3.2.2 Containerization Overheads To account for the overhead of containerization, we also ran ReactDB while submitting empty transactions with concurrency control disabled. We observe roughly constant overhead per transaction invocation across scale factors of around 22 μsec. Even though for the setup with TPC-C this overhead would correspond to close to 18%, we measured that thread switching overhead between the worker and transaction executor across different cores is a largely dominant factor and is dependent on the machine used. When compared with executing the TPC-C application code directly within the database kernel without any process separation, as in Silo, the overhead is significant, but if a database engine with kernel thread separation is assumed, as is the normal case, the overhead is negligible.

3.4.3.3 Effect of Load

To further drill down on the potential benefits of asynchronicity under concurrency, we evaluate in this section the two database architectures shared-nothing-async

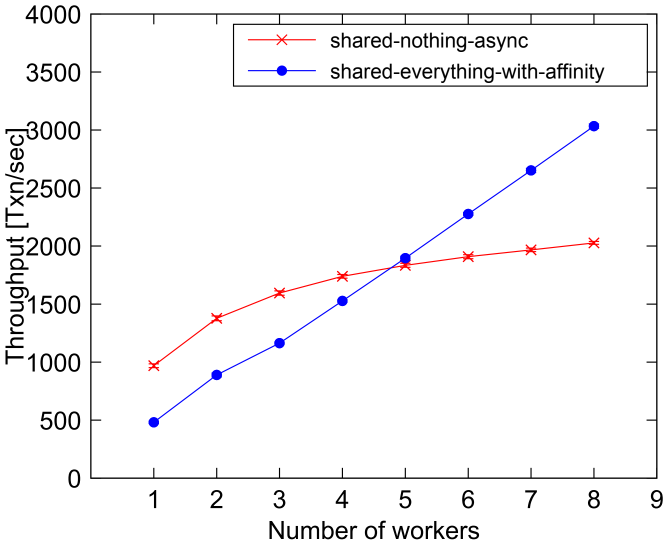


Figure 3.14: Throughput of new-order-delay transactions with varying load.

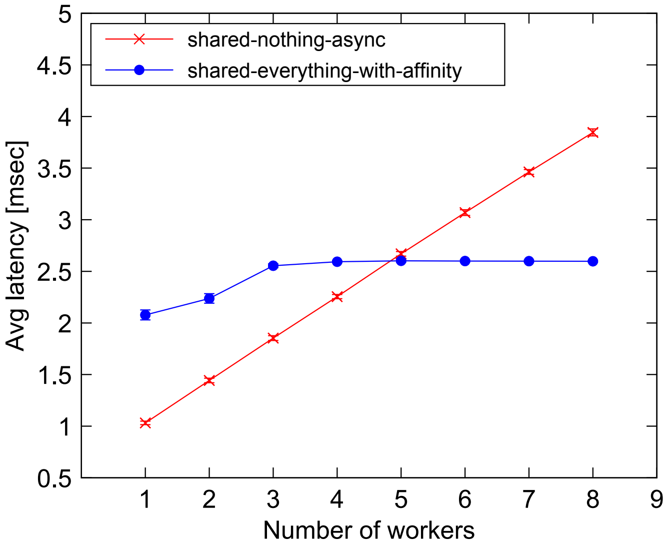


Figure 3.15: Latency of new-order-delay transactions with varying load.

and shared-everything-with-affinity under varying load. We control the amount of load by varying the number of workers from 1 to 8, while keeping the number of warehouses constant at a scale factor of 8. For clarity, we focus exclusively on new-order transactions. Each new-order consists of between 5-15 items and we force each of the items to be drawn from a remote warehouse with equal probability. Since the default new-order formulation has limited parallelism in the logic executed at remote warehouses, we augmented the logic for stock data update with an artificial delay between 300 and 400 μsec by generating random numbers to model stock replenishment calculations. This increases the overall work in the transaction without increasing its footprint and the contention on the database.

Figures 3.14 and 3.15 show the throughput and latency, respectively, of running 100% new-order-delay transactions under increasing load. With one worker, the throughput of shared-nothing-async is double that of shared-everything-with-affinity. The former executes all the stock updates across 5-6 remote warehouse asynchronously (average distinct remote warehouses chosen from 7 using a uniform distribution) fully utilizing the available hardware parallelism, while the latter executes the entire transaction logic sequentially. Although shared-nothing-async incurs higher communication cost in dispatching the stock updates to be performed by different warehouse reactors, the greater amount of work in each stock update makes it worthwhile in comparison to sequential shared memory accesses in shared-everything-with-affinity. Conversely, as we increase the number of workers and thus pressure on resources, the throughput of shared-nothing-async starts growing less than that of shared-everything-with-affinity. Note that the abort rate for the deployments was negligible (0.03-0.07%), highlighting the limited amount of contention on actual items.

In summary, these results suggest that the most effective database architecture may change depending on load conditions when asynchronicity can be exploited by transaction code. Under high load, shared-everything-with-affinity exhibits the best performance among the architectures evaluated, since it reduces overhead at the expense of not utilizing at all intra-transaction parallelism. On the other hand, when load conditions are light to normal and when transaction logic comprises enough parallelism, shared-nothing-async can achieve substantially higher throughput and lower latency. To further validate these observations, we evaluate in Section 3.4.3.4 the effects of varying cross-reactor accesses in the TPC-C benchmark under conditions of high load.

3.4.3.4 Effect of Cross-Reactor Transactions

In this section, we evaluate the impact of cross-reactor transactions on the performance of the different database architectures, complementing the results of Section 3.4.3.3. For clarity, we focus exclusively on new-order transactions. To vary the percentage of cross-reactor new-order transactions, we vary the probability that a single item in the transaction is drawn from a remote warehouse (the

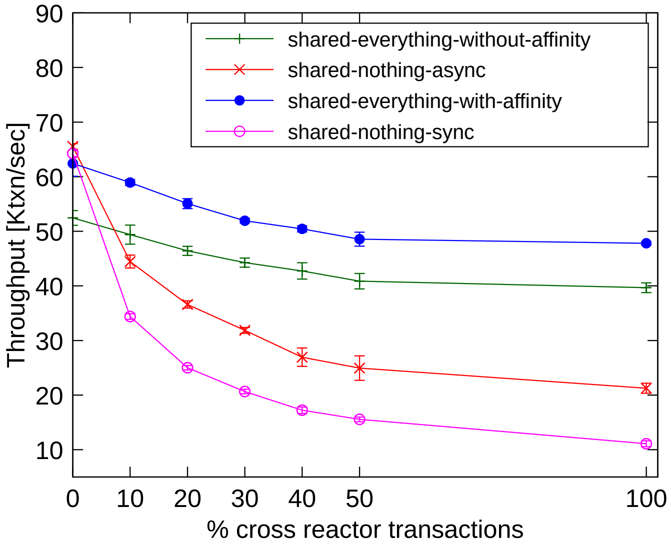


Figure 3.16: Throughput of cross-reactor new-order transactions.

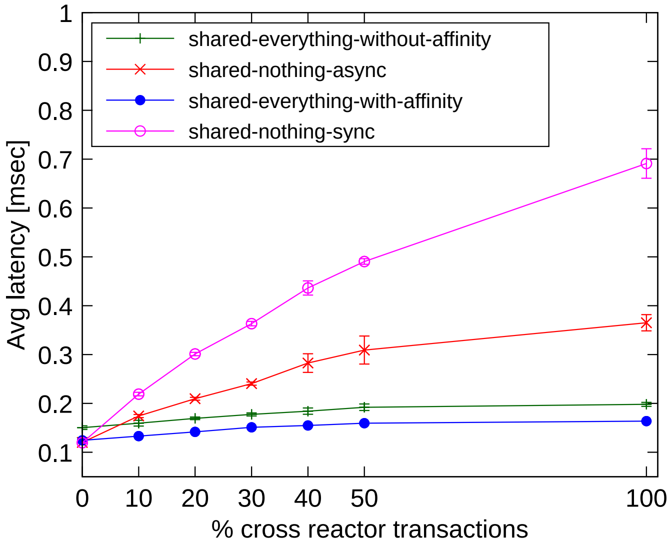


Figure 3.17: Latency of cross-reactor new-order transactions.

remote warehouses are again chosen with an equal probability). Each new-order consists of between 5-15 items. We do *not* extend the new-order transaction with any additional computations, limiting the amount of intra-transaction parallelism available.

Figures 3.16 and 3.17 show the throughput and latency, respectively, of running the TPC-C benchmark with 100% new-order transactions at a scale factor of eight, i.e., eight warehouses receive transactions from eight workers at peak load. Since ReactDB uses Silo's OCC protocol and owing to the low contention in TPC-C even upon increasing the number of remote items, we would expect the throughput and latency of shared-everything-with-affinity to be agnostic to changes in the proportion of cross-reactor transaction as per the results in [125]. However, we see a gradual decrease in the throughput and an increase in the latency for all the deployments. We believe these effects are a consequence of cache coherence overheads and remote memory latency in the NUMA machine employed. We observe further that both shared-nothing-sync and shared-nothing-async configurations exhibit the same latency and throughput at 0% cross-reactor transactions as shared-everything-with-affinity. However, there is a sharp drop in the performance of shared-nothing deployments from 0% to 10% cross-reactor transactions. This effect is in line with our previous observation that sub-transaction invocations require expensive migration of control in contrast to both shared-everything-without-affinity and shared-everything-with-affinity. Note that the abort rate for all the different deployments remained negligible (0.02%-0.04%), highlighting the limited amount of contention on actual items.

We observe that shared-nothing-async exhibits higher resilience to increase in cross-reactor transactions when compared with shared-nothing-sync. Both latency and throughput of shared-nothing-async are better by roughly a factor of two at 100% cross-reactor transactions. This is because shared-nothing-async employs new-order transactions with asynchronous sub-transaction invocations on remote warehouse reactors, and tries to overlap remote sub-transaction invocation with execution of logic locally on a warehouse reactor. This demonstrates how application programs can leverage the programming model to engineer application code using reactors with different performance characteristics. At the same time, infrastructure engineers can select the database architecture that best fits the execution conditions for the workload without changes to application code. In the case of peak load and limited intra-transaction parallelism, `shared-everything-with-affinity` turned out to be the best architecture among the ones considered for this scenario, in line with the results of [125].

3.5 Related Work

3.5.1 In-memory OLTP Databases

H-Store [121] and HyPer [80] follow an extreme shared-nothing design by having single-threaded execution engines responsible for each data partition. As a result, single-partition transactions are extremely fast, but multi-partition transactions and skew greatly affect system throughput. LADS [133] improves upon this limitation by merging transaction logic and eliminating multi-partition synchronization through dynamic analysis of batches of specific transaction classes. In contrast to these shared-nothing engines, shared-everything lock-based OLTP systems specifically designed for multi-cores, such as DORA [98] and PLP [99], advocate partitioning of internal engine data structures for scalability. Orthrus [106] partitions only the lock manager and utilizes a message-passing design for lock acquisition to reduce lock contention across multiple cores for contended workloads. In contrast to the baked-in architectural approach of earlier engines, REACTDB borrows the highly-scalable OCC implementation of Silo [125], building on top of it a virtualization layer that allows for flexible architectural deployments, e.g., as a classic shared-everything engine, a shared-nothing engine, or an affinity-based shared-everything engine. In addition, REACTDB is not restricted to specific transaction classes, supporting transactions with, e.g., user-defined aborts, conditionals, and range queries. Finally, REACTDB is the first engine realizing the programming model of reactors, which provides developers of latency-sensitive OLTP applications with a new abstraction to reason about latency in transactional programs. The reactor programming model and REACTDB are a refinement and concretization of our earlier vision of transactional partitioning [111].

3.5.2 Transactional Partitioned Data Stores

A class of systems provides transactional support over key-value stores as long as keys are co-located in the same machine or key group [42, 43]. Warp [58], in contrast, provides full transaction support with nested transactions, but limits query capabilities, e.g., no predicate reads are provided nor relational query support. The limited transactional support and low-level storage-based programming model make it difficult to express OLTP applications as opposed to the reactor programming model, which provides serializable transactions with relational query capabilities. Recent work has also focused on enhancing concurrency through static analysis of transaction programs [91, 136]. The latter work could be assimilated in the implementation of REACTDB's concurrency control layers as future work.

3.5.3 Asynchronous Programming

As mentioned previously, reactors are a novel restructuring in the context of databases of the actor model [3]. In contrast to regular actors, reactors comprise an explicit memory model with transactions and relational querying, substantially simplifying program logic. These features make the reactor model differ significantly from the virtual actors of Orleans [23] and from other actor-based frameworks [14, 70]. Recent work in Orleans has focused on a vision of integrating traditional data-management functionality in a virtual actor runtime for the middle tier of a classic three-tier architecture [26, 54]. This approach is complementary to our work of integrating actor features in a database system, i.e., enriching the data tier itself. Additionally, REACTDB comprises building a high-performance, scalable, multi-core OLTP system with an actor-oriented programming model and latency control, which is not the target design and feature set of the vision for Orleans [26].

As explained in Section 3.2.2, reactors are related to the early work on Argus [88] because of the asynchronous transactional programming model supporting nested function calls; however, the reactor programming model is substantially different from that of Argus. First, the use of a relational data and query model is a central idea of reactors, but not of Argus. Note that the latter is not a simple restriction of the former, because the programming issues handled by a relational abstraction, e.g., physical data independence, would need to be coded from scratch at a very low level in Argus. Second, user-defined logical actors are a central idea of reactors, but not of Argus either. A process in Argus is a low-level thread of control mapped to physical execution, while reactors are *logical* computational entities, which do not map one-to-one to physical threads in ReactDB. Third, reasoning about latency from the programming model is a central idea of reactors, but again not of Argus. Even though Argus has low-level asynchronous calls, it lacks an explicit cost model of synchronous and asynchronous communication. In addition, on the system implementation level, ReactDB is an OLTP database system designed for low-overhead virtualization of database architecture, which was never the focus of Argus. These differences to Argus also distinguish our work from a large class of object-oriented distributed computing and operating systems [30, 40, 44, 84, 95].

3.5.4 Database Virtualization

Virtualization of database engines for cloud computing has focused on particular target database architectures, e.g., shared-nothing databases with transactional support only within partitions [25] or distributed control architectures with weaker consistency guarantees [81]. By contrast, ReactDB offers infrastructure engineers the possibility to configure database architecture itself by containerization, while maintaining a high degree of transaction isolation. Our results support recent observations of low overhead of use of container mechanisms together with an in-memory database [92], while showing that even more flexibility in database architecture can be achieved at negligible cost.

The design of ReactDB is reminiscent of work on OLTP on hardware islands [104] and on the cost of synchronization primitives [45]. Our work provides a solution for low-level control of the effects of hardware heterogeneity through a new programming model and system design allowing reasoning about latency at a high level.

3.6 Conclusion

In this chapter, we introduced reactors, a new relational abstraction for in-memory OLTP databases. This abstraction exposes an asynchronous programming model allowing for flexible program design and reasoning about latency while maintaining serializability of transactions. We presented the design of ReactDB, the first implementation of reactors. ReactDB allows for flexible and controllable database architecture configuration at deployment time. Reactors open up a variety of directions for future work, ranging from reactor database modeling to efficient mapping of reactors to distributed hardware architectures.

Chapter 4

An Evaluation of Intra-Transaction Parallelism In Actor Database Systems

> *"No amount of experimentation can ever prove me right; a single experiment can prove me wrong."*
>
> — Albert Einstein

Over the past decade, we have witnessed dramatic evolution in hardware capabilities and database system designs. The new trends in Moore's law forecast a scenario in which processors with increasing core counts keep getting cheaper. Similarly, main-memory capacity is increasing while cost per gigabyte dropping. As a result, a single-node multi-core machine today can be superior in performance and price than a cluster of machines two decades back. At the same time, a recent vision of actor database systems has been proposed to increase the programmability of database systems by fusing actor programming models with state management capabilities. These new systems promise to provide support for upcoming latency-sensitive applications with complex application logic and task-level parallelism. In this scenario, a natural question is whether an actor database system with an asynchronous programming model can adequately expose the parallelism available in modern multi-core hardware. Towards that aim, we conduct in this chapter a thorough evaluation of the factors that affect intra-transaction parallelism in a prototype actor database system, named REACTDB, for an application benchmark designed with task-level parallelism in mind and running on a multi-core machine. Based on our evaluation, we observe that the mechanisms employed to implement transactional parallelism in REACTDB introduce minimal overhead, indicating that expectations regarding parallel performance resulting from the use of asynchronicity constructs can be verified in practice. As a consequence, given the hardware landscape today, we believe it is time for the database community to carefully consider intra-transaction parallelism in benchmarks with complex application logic in addition to only inter-transaction parallelism and its associated trade-offs.

4.1 Introduction

The past decade has seen a revolution in the hardware landscape and consequently in database system design. As the gains of single-threaded performance of processors dwindled, chip designers have responded by adding processing power in the form of more cores in a processor. In this new continuation of Moore's law, processors with growing numbers of cores become increasingly affordable over time. This trend has also been witnessed in the capacity growth and price decline of main-memory, thus making a single-node multi-core machine today more powerful and cheaper than a cluster of machines two decades back. This has brought focus back to overhead and scalability issues in the design of traditional database systems for this brave new world of large main memory and multi-core machines [76, 121]. The database system community has responded by redesigning the architecture and internals of database systems for this new scenario [50, 78, 80, 83, 98, 125]. The guiding ambition for this response, particularly in OLTP DBMS, has been on improving transactional scaleup or high-volume processing of sequential transactions.

A recent radical vision of actor database systems has proposed integration of actor-oriented programming models with the traditional database system features of transactions and declarative querying (Chapter 2). The vision argues for the use of a logical actor programming model (1) to model applications in terms of actors for modularity benefits and (2) to leverage asynchronicity in the programming model to specify available task parallelism in the application logic. At the same time, the actor database system guarantees that the execution of application logic is carried out under transactional guarantees, thus freeing application developers to write parallel programs without having to reason about complex synchronization for correctness. The vision thus argues for marrying actor programming features with the state management capabilities of database systems to enable application developers to reap the benefit of application-level modularity and asynchronicity while maintaining their insulation from concurrency and failure issues.

Promising results have been shown in this earlier vision in leveraging asynchronicity in transactions for the proposed SmartMart benchmark (Section 2.6 in Chapter 2), which models a simplified future IoT supermarket application for next-generation self-checkout [11, 103]. The benchmark attempts to model a new and upcoming class of latency-sensitive applications with complex application logic mixed with data accesses. Examples of complex application logic can range across domains such as financial computations [33], real-time simulations [127, 130, 131], and online machine learning methods [90], to name a few. In the benchmark, the complex application logic is modeled through a mix of read-mostly transactions [128] with calculations for mean and standard deviation intermixed with conditional statements and aggregation operations. The benchmark aims to model application logic with an amount of parallelizable work and task-level parallelism sufficient to benefit from parallel execution.

In this work, we aim at revisiting in the context of actor database systems and multi-core machines the classical performance studies from two decades back [37,48] that evaluated performance benefits from parallelism in distributed database systems [49]. We evaluate whether the promise of parallel programming in an actor database system, namely REACTDB (Chapter 3), actually translates to observable performance benefits in a single-node, multi-core machine today. In our study, we do not focus on the effects of concurrency control mechanisms on transaction processing performance that have been studied recently [67,135], but solely on the factors that affect parallel programs and how their effects manifest in an actor database system. To the best of our knowledge, this is the first evaluation of intra-transaction parallelism benefits in actor database systems for multi-core machines.

In our evaluation, we observe that the effect of classic factors affecting parallelism clearly manifest in the measured behavior for intra-transaction parallelism in the studied benchmark. The results indicate that an actor database system can implement transactional parallelism with minimal overhead, and in a way that allows application developers to exploit such parallelism in modern multi-core machines. Based on our evaluation, we believe it is time for the database community to investigate intra-transaction parallelism and its trade-offs with inter-transaction parallelism in complex application benchmarks as opposed to focusing only on scalability of sequential transactions and associated microarchitectural effects.

In summary, we make the following contributions in this chapter:

1. We perform a thorough study of the factors that affect parallelism and their effects on a benchmark with complex application logic in the REACTDB actor database system prototype.

2. We discuss in detail the implementation of internal components that facilitate transactional parallelism in REACTDB and illustrate through our experiments that the associated overhead introduced in the benchmark studied is minimal.

The remainder of this paper is organized as follows. In Section 4.2, we present briefly the classical factors that affect parallel program performance as well as the actor database system REACTDB and the benchmark SmartMart used in our evaluation. In Section 4.3, we drill down into the design and implementation features of REACTDB that ensure the benefits of intra-transaction parallelism in application programs are maintained by the system. In Section 4.4, we evaluate the system by varying the benchmark parameters and configurations to quantify the effects of the parallel programming factors enunciated earlier. Finally, we discuss related evaluation work in Section 4.5 and conclude.

4.2 Background

In this section, we first review the classical factors that affect performance of parallel programs, whose effects we evaluate in the context of an actor database system (Section 4.2.1). We then provide background on actor database systems and the particular system chosen for our experimental evaluation, namely REACTDB(Section 4.2.2). Finally, we briefly describe the benchmark employed in this chapter, SmartMart (Section 4.2.3).

4.2.1 Factors Affecting Parallelism

There are several classic factors that affect the performance of parallel programs, the main of which we recap below.

4.2.1.1 Overhead

Any parallel program is affected by the cost paid by its execution overheads. Examples of overhead comprise: (1) *Startup costs* that a program has to pay to initiate the parallel computations, including the cost of communication with the parallel processing units necessary to dispatch the work to be performed; (2) *Transactional overheads* that a program has to pay while running in a database context to obtain transactional guarantees. Some of the overheads can be potentially parallelized, e.g., commit protocol, transaction management overheads, communication cost to send results back from the parallel processing units, while others cannot, e.g., communication cost to initiate parallel computations.

4.2.1.2 Parallelizable Work and Dependencies

Any parallel program is governed by Amdahl's law [12], which prescribes that the performance of a parallel program is governed by its amount of parallelizable work, i.e., the fraction of total work that would benefit from parallel execution. For a program having a small amount of parallelizable work, the gains from parallelism with increasing numbers of parallel resources will quickly diminish. As a result, the nature of the program and its functional decomposition, including dependencies among tasks, has a direct influence on the parallelism benefits that can be achieved. In particular, parallelism will not benefit a sequential program, while an embarrassingly parallel program can expect almost linear improvements in performance from corresponding increases in parallel resources. Note that dependencies have a direct influence on the synchronization that must be exercised in the program, leading to communication overheads.

4.2.1.3 Interference

Typically, parallel programs are written without consideration for the amount or utilization level of parallel processing resources. In practice, however, when resources are shared for the processing of parallel programs, slowdowns can occur. Resource sharing can happen due to (1) execution of multiple parallel programs at the same time, or (2) lack of available resources to sustain the amount of parallelism specified in the program.

4.2.1.4 Skew

In order to get the maximum gain from parallelism, a parallel program must keep all the parallel resources busy at all times. This requires that all tasks executed on the parallel resources have almost the same processing time. This turns out to be especially important as parallel resources are added, since the execution time of the parallel program can end up being dominated by its slowest task. As a result, any imbalance, i.e., skew, in the execution time of the tasks of a parallel program can negatively affect its gains from parallelism.

4.2.2 Actor Database Systems and ReactDB

Actor programming models [4] and runtimes [6, 56, 110] have seen increased adoption of late to program the middle tier of stateful applications [119]. Concomitantly, a greater need for state management features in such systems has emerged. To partially address this need in the middle tier, the Orleans project has introduced the notion of virtual actors [23], as well as initiated efforts to integrate actor indexing [26] and transaction support [54]. However, this line of work has largely targeted keeping the single-threaded actor programming model intact and trading availability for consistency for deployment in a cloud computing infrastructure.

Recently, a vision for integrating actor programming models and database systems at the data tier has been outlined (Chapter 2). This work advocates bringing the benefits of modularity, performance, and security from actors to classic database systems. Within this context, the possibility of exploiting parallelism through asynchronicity of communication among actors while maintaining transactional guarantees arises. Even though prior work in actor database systems has shown promising performance results in the use of asynchronicity in transactions (Section 2.6.6.2 in Chapter 2), the effect of the multiple factors affecting parallelism still needs to be thoroughly explored in these systems.

Towards the latter, we base this study on ReactDB, an in-memory actor database system designed for multi-cores that provides a logical, actor-oriented programming model (Chapter 3). In ReactDB, a logical actor is referred to as a *reactor*, because the state of an actor is abstracted using relations. Reactors are purely an application-defined construct and exist for the lifetime of the application. Communication among reactors is achieved by asynchronous nested function invocation semantics. Concretely, function invocations on reactors are asynchronous calls returning promises [89], referred in the remainder as *futures*, and can cause nested function invocations on other reactors. However, any function invocation on a reactor is guaranteed to be atomic and serializable. ReactDB does not at present provide durability guarantees. Moreover, application programs in ReactDB are written using the reactor programming model directly in C++, where the state of reactors is abstracted through indices supporting interactions against a record-manager interface. Thus, according to the classification in the actor database systems manifesto (Section 2.4 in Chapter 2), REACTDB only supports Features 1-4. Despite these limitations, the feature set offered by REACTDB is sufficient for evaluation of the potential benefits of intra-transaction parallelism achieved through asynchronicity in an actor database system, which is the focus of this chapter.

REACTDB has been designed to allow flexible specification at deployment time of underlying database architecture across the extremes of shared-nothing and shared-everything without necessitating any changes to application code. To achieve this flexibility, REACTDB virtualizes database architecture by abstracting the notions of memory and computational resources. To abstract memory, REACTDB introduces *containers*, which are shared-memory regions under a concurrency control protocol. Currently, REACTDB utilizes the optimistic concurrency control (OCC) implementation of Silo [125]. To abstract computational resources, REACTDB assigns *transaction executors* to containers. A transaction executor is implemented by a thread pool, pinned to a core, employing a cooperative scheduling strategy. To create a deployment, REACTDB requires the specification of the mapping of transaction executors to containers and reactors to transaction executors, such that a reactor can be mapped to only one container. In case a reactor is mapped to more than one transaction executor in a given container, a *transaction router* selects the target destination executor for a function call based on a user-configured policy, e.g., affinity-based or round-robin. We utilized this flexible deployment feature of containers in our experimental evaluation in Section 4.4 while keeping the application programs unchanged. We discuss some of the implementation features of REACTDB that affect intra-transaction parallelism performance in Section 4.3.

4.2.3 SmartMart Benchmark Description

The SmartMart benchmark (Section 2.6 in Chapter 2) was designed to model a simplified future IoT supermarket application for next-generation self-checkout [11, 103]. The application models the workflow of a customer inside the supermarket carrying a smart shopping cart that is equipped with sensors to itemize its physical contents.

The benchmark consists of seven relations, while the workload comprises of a workflow of (1) adding items and (2) checkout, simulating the actions of customers in the supermarket. The throughput and latency metrics are reported for the entire workflow. In the benchmark specification (Section 2.6 in Chapter 2), the functional decomposition of the application across actors has already been done along with the schema specification for each actor and the declarative queries to interact with the actor state. We used the same decomposition of actors in our experimental evaluation in Section 4.4 and ported the declarative queries into handcrafted physical plans over the database index interfaces of REACTDB.

Importantly, most of the asynchronicity in the transactions in the benchmark arises from the functionality to calculate various types of discounts. In particular, the application defines two different types of discounts on each item in the inventory: (1) fixed discount and (2) variable discount. The fixed discount is customized based on the marketing group of the customer, while the variable discount is computed based on the demand for the item over a pre-defined window. The price and fixed discounts are fetched asynchronously when items are added to the cart, while the variable discount is only asynchronously computed during checkout according to a formula based on the mean and standard deviation of item quantities ordered in a history window (Section 2.6 in Chapter 2). At checkout, the entire price and discounts are aggregated for the customer's order.

We employ the SmartMart benchmark because it has the necessary parameters that allows us to vary and observe the effects of the factors outlined in Section 4.2.1. Moreover, the benchmark's fit to actor database systems facilitates the evaluation of intra-transaction parallelism in such systems, as is the focus of the present study.

4.3 Transactional Parallelism Implementation

In this section, we discuss the implementation details of REACTDB related to transactional parallelism. REACTDB's core abstraction for intra-transaction parallelism is that of asynchronicity in nested function calls, which are provided while maintaining transactional guarantees of atomicity and serializability. We begin by first discussing the thread pool management features in a transaction executor, which are the active computational entities in REACTDB (Section 4.3.1). We then explain the workflow followed by a transaction executor (Section 4.3.2), and continue by discussing the implementation of a function calls on reactors and the associated dispatch mechanism across containers and transaction executors (Section 4.3.3). Finally, we outline the commit protocol employed by the system (Section 4.3.4), before discussing issues of code generation and memory management that help in achieving high performance in REACTDB (Section 4.3.5).

4.3.1 Thread Management In Transaction Executors

A transaction executor is a thread pool pinned to an actual physical core. Transaction executors are created when REACTDB is bootstrapped based on an assignment to containers specified in configuration files. REACTDB runs as a single process, and all the threads in a transaction executor after creation wait to be scheduled in order to run. Every transaction executor has its own thread scheduler, which is configured with the number of active threads allowed to run at any point of time. Figure 4.1 depicts the thread states and the events that cause transitions between these states. Every thread in a transaction executor is in one of these four states at any point of time.

After creation, all the threads are in the READY state and wait for permission from the scheduler to run. A scheduler allows a thread to run only if the number of current active threads does not exceed a limit of active threads configured per scheduler. Once a thread gets permission to run, it enters the ACTIVE state and keeps dequeuing and executing (sub-)transaction requests from the transaction

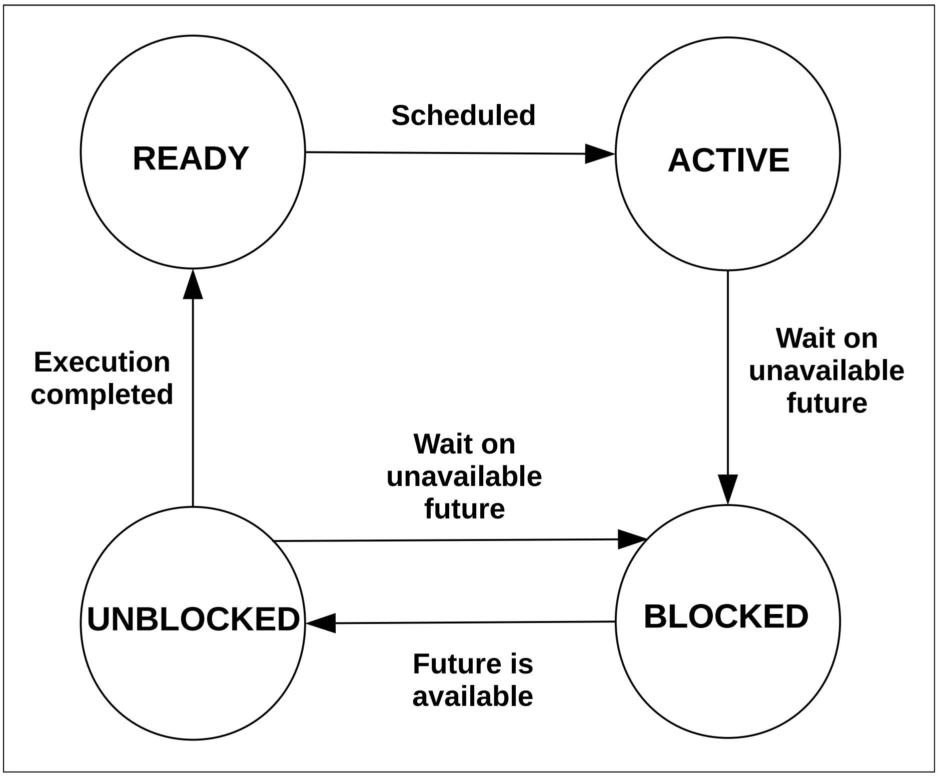


Figure 4.1: Transaction Executor Thread States and Transitions

executor queue. During the course of execution of (sub-)transactions, the thread can get blocked if it needs to wait for the availability of a future result of a nested sub-transaction. In such a case, the thread transitions to the BLOCKED state and notifies the scheduler. The scheduler then permits another thread in READY state to begin execution, if available. The BLOCKED thread transitions to the UNBLOCKED state once the sub-transaction result it is waiting for becomes available. Until the end of the execution of the current (sub-)transaction, the thread never returns to the ACTIVE state, but can transition back and forth between BLOCKED and UNBLOCKED. When the entire execution of the (sub-)transaction is complete, the thread returns back to the thread pool, transitioning to the READY state. As such, the thread can now once again wait for permission from the scheduler to run.

These mechanics of thread scheduling aim to ensure that the queue of a transaction executor is continuously drained by active threads up to the limit on thread resources. This minimizes delay on scheduling of transactions and sub-transactions while controlling admission to compute resources allocated to the system. Note that a thread in the ACTIVE state that never needs to wait for an unavailable sub-transaction result will drain the transaction executor queue continuously and not return back to the thread pool.

Algorithm 4.1 Transaction Executor Workflow

```
 1: procedure TxnExecutor::executeForever (sched, queue, coord, container)
 2:     while $true$ do
 3:         $sched.await()$
 4:         while $runningUnblocked$ do
 5:             $txn \leftarrow queue.deque()$
 6:             if $txn.isRootTxn()$ then
 7:                 $tid \leftarrow generateTid()$
 8:                 $txn.setTid(tid)$
 9:                 $dbCtx \leftarrow container.createDbCtx(tid)$
10:             else
11:                 $dbCtx \leftarrow container.getOrCreateDbCtx(txn.getTid())$
12:             end if
13:             $txn.setDbCtx(dbCtx)$
14:             $txn.run()$
15:             $success \leftarrow txn.waitForSubTxns()$
16:             if $txn.isRootTxn()$ then
17:                 $coord.commitOrAbort(txn, container, success)$
18:             else
19:                 $txn.dispatchResult();$
20:             end if
21:         end while
22:     end while
23: end procedure
```

4.3.2 Transaction Executor Workflow

Algorithm 4.1 represents the workflow of the threads in the transaction executor. The procedure EXECUTEFOREVER receives as arguments the scheduler for the transaction executor, the transaction executor queue, the container to which the transaction executor belongs, and the transaction coordinator for the container. As explained in the previous section, when the threads in a transaction executor are created, they wait for permission from the scheduler to run (line 3). Once permission is granted by the scheduler, the thread dequeues (sub-)transactions from the transaction executor queue (line 5). Every transaction has a database context that stores necessary concurrency control information such as write and read sets for OCC. This database context is stored in the container, since the context needs to be re-used across any nested sub-transaction executions touching the same container.

Every database context of a transaction is mapped by a transaction identifier that is generated once when a root transaction begins execution. Even though this identifier is carried along in sub-transaction invocations across containers, every transaction has only one database context in any given container. Lines 6-12 show the creation of transaction identifier, the creation of the database context for a transaction, and the lookup of database contexts for sub-transactions. To avoid conflicts among sub-transactions of the same root transaction, the transaction executor also ensures that a database context in a container is accessed by only one thread at any point of time and conservatively aborts any transactions violating this requirement. The database context is utilized during the actual execution of (sub-)transaction code (line 14), where the run method abstracts the application logic.

Once the application logic completes, an explicit synchronization is done to wait for the completion of nested children sub-transactions. This guarantees that the parent can only return once all child sub-transactions are complete, recursively (line 15). As such, when all children of a root transaction complete, the transaction execution is complete and a commit protocol must be initiated based on the results of the transaction execution. The commit protocol runs in the transaction coordinator, and ensures that a transaction is aborted if the execution encountered errors and committed otherwise (line 17). Finally, if the transaction is not a root transaction, the result of the sub-transaction must be dispatched back to the caller (line 19). A transaction executor also performs other functionality for memory management and garbage collection that we omit from the pseudocode for simplicity.

The workflow followed by a transaction executor aims to ensure that transactional contexts are managed properly across multiple containers in support of atomicity and serializability. Note that the workflow does not make any assumptions on patterns of nested asynchronous function calls in application logic and will accept programs with arbitrary nested invocations of functions and synchronization structures, allowing for expression of complex intra-transaction parallelism in application control flow.

4.3.3 Asynchronous Function Call Implementation

Having outlined the workflow of a transaction executor in the previous section, we now explain how asynchronous function calls are implemented in ReactDB. Specifically, we outline the steps triggered when the application logic invokes a function call in a particular reactor.[1]

Algorithm 4.2 shows the corresponding pseudocode. The function EXEC captures the dispatch logic of a function call on a reactor, abstracted in ReactDB by a (sub-)transaction txn. EXEC receives as arguments a handle to the parent (sub-)transaction from which the (sub-)transaction was invoked (empty for a root transaction), the container where the parent transaction was executing, the reactor on which the (sub-)transaction is invoked, and the arguments to the (sub-)transaction. Lines 2 and 3 store the (sub-)transaction inputs and the (sub-)transaction reactor inside txn so that these values can be later accessed by the

[1] In [113], such a call corresponds to the use of the construct fn(args) on reactor X in the application logic.

Algorithm 4.2 Asynchronous Transaction Execution

```
 1: function TXN::EXEC (parentTxn, txn, srcContainer, reactor, args)
 2:     $txn.setInput(args)$
 3:     $txn.setReactor(reactor)$
 4:     $result \leftarrow txn.createResultFuture()$
 5:     if $\neg txn.isRootTxn()$ then
 6:         $parentTxn.addToSubTxns(txn)$
 7:         $txn.setTid(parentTxn.getTid())$
 8:     end if
 9:     $dstCont \leftarrow srcContainer.GetContainer(reactor)$
10:     if $dstCont = srcContainer$ then
11:         $txn.setDbCtx(parentTxn.getDbCtx())$
12:         $txn.run()$
13:     else
14:         $txn.setContainer(dstCont)$
15:         $dstContainer.schedule(txn)$
16:     end if
17:     return $result$
18: end function
```

transaction logic. Line 4 creates the future result of the transaction that will be later returned at the end of the function (line 17). If `txn` is a not a root transaction, then its handle is stored in the list of sub-transactions of the parent (lines 5 and 6). This list of sub-transactions is used by the parent to ensure synchronization of its children sub-transaction executions in `waitForSubTxns` (Algorithm 4.1). The identifier of the sub-transaction is also set to that of the parent (sub-)transaction for all non-root transactions (line 7).

To execute the sub-transaction, the destination container mapped to the `reactor` is looked up by consulting the reactor-to-container mapping stored in the source container (line 9). If the destination container is the same as the source container (only true for non-root transactions), then the database context of the parent sub-transaction is stored in the child sub-transaction and the method call to the application logic is directly invoked. This action chains the invocation sequence, leading to synchronous execution (lines 10-12). The intuition for this decision is that we wish to eliminate the overhead of rescheduling the sub-transaction when a migration to another container is unnecessary. As a special case, this decision also ensures that a sub-transaction invoked on the same reactor is synchronously executed with minimal overhead.

If the source container differs from the destination container, the destination container is stored in the (sub-)transaction for later use during the commit protocol. Then, the transport driver of the source container is invoked to move the `txn` to the destination container (line 15). In the current implementation of REACTDB, the logic utilizes shared-memory access to directly invoke `schedule` on the destination

Algorithm 4.3 Commit Protocol

```
procedure TxnCoordinator::CommitOrAbort(rootTxn, srcContainer, commit)
    if $commit$ then
        $success \leftarrow srcContainer.getDbCtx().validate()$
    else
        $success \leftarrow false$
    end if
    $containers \leftarrow rootTxn.getRemoteContainers(srcContainer)$
    if $success$ then
        for all $container \in containers$ do
            $success \leftarrow container.getDbCtx().validate()$
            if $\neg success$ then
                break
            end if
        end for
    end if
    if $success$ then
        $srcContainer.getDbCtx().write()$
        for all $container \in containers$ do
            $container.getDbCtx().write()$
        end for
    else
        $srcContainer.getDbCtx().abort()$
        for all $container \in containers$ do
            $container.getDbCtx().abort()$
        end for
    end if
    $rootTxn.setCommitStatus(success)$
    $rootTxn.dispatchResult()$
    $rootTxn.end()$
end procedure
```

container, which looks up the destination transaction executor for the reactor with the aid of the transaction router and enqueues the (sub-)transaction into the corresponding transaction executor queue.

4.3.4 Commit Protocol

After the execution of the transaction logic is completed, ReactDB ensures that all the results of the sub-transactions of a root transaction are available. Then, the transaction coordinator initiates a linear two-phase commit (2PC) protocol to either commit or abort the root transaction. Algorithm 4.3 outlines the implementation of the 2PC protocol in ReactDB. The validation phase of the OCC protocol is executed first in the source container, i.e., where the root transaction was initiated and consequently the transaction coordinator was invoked (line 3). After this validation step, all the remote containers that are spanned by the transaction are looked up recursively through the chain of sub-transactions starting from the root transaction (line 7).

If validation was successful in the source container, lines 8-15 proceed to run the validation phase of the OCC protocol on each of the remote containers. Following this process, either the write phase (lines 17-20) or the abort phase (lines 22-25) is executed, depending on the combined result of validation. At the end, the commit status of the root transaction is set, so that it can be retrieved by calling code (line 27). At this point, the execution and commitment of the transaction is complete, so the caller is notified of the transaction completion (line 28) and cleanups are performed for the transaction (line 29).

4.3.5 Other Implementation Factors

In this section, we highlight additional implementation decisions taken during the design of ReactDB that contribute to low overhead and high performance, thus accentuating the benefits of intra-transaction parallelism.

4.3.5.1 Efficient Code Generation

ReactDB has been designed as a framework heavily using C++11 templates to minimize any dynamic dispatches during execution. In particular, C++ templates are used in implementation throughout the entire code line, including index structures, concurrency control mechanisms, transaction coordinators, transaction executors, transaction routers, schedulers and containers. The application code in stored procedures extends ReactDB transaction classes and is thus compiled with ReactDB instead of being linked separately, providing the compiler with a large body of program code to optimize. In addition, the compiler has the opportunity to specialize the code generation according to a set of static configuration options of ReactDB. Post compilation, application binaries for the Smallbank [9, 64], SmartMart [112], and TPC-C benchmarks [123] reach a size of 34 MB, 32 MB and 41 MB, respectively.

4.3.5.2 Efficient Memory Management

In order to prevent bottlenecks in memory allocation, ReactDB uses a custom memory allocator for each thread in the transaction executor that pre-allocates memory on the heap and only dynamically allocates memory if pre-allocated memory is exhausted. Furthermore, any memory dynamically allocated is not freed immediately, but reserved by the allocator for later use. Thus, this results in the memory pool being resized dynamically. Transaction execution consumes memory from the memory pool while garbage collection reclaims memory back on completion of transactions. REACTDB also makes extensive use of the stack instead of the heap where possible in its implementation.

4.4 Evaluation

In this section, we evaluate the effects of classical parallelism factors as outlined in Section 4.2.1 on intra-transaction parallelism in actor database systems. More concretely, we aim at answering the following questions:

- How does the amount of parallelizable work and the overheads of an actor database system affect the overall speedup of transaction programs (Sections 4.4.2 and 4.4.3)?
- How does interference from inter-transaction parallelism affect the benefits of intra-transaction parallelism (Section 4.4.4)?
- How does skew or load imbalance in intra-transaction parallelism affect the benefits experienced by transaction programs (Section 4.4.5)?

4.4.1 Experimental Setup

4.4.1.1 Hardware

We employ a machine with two sockets, each with one eight-core 2.6 GHz Intel Xeon E5-2650 v2 processor with two physical threads per core, leading to a total of 32 hardware threads. Each physical core has a private 32 KB L1 data and instruction cache and a private 256 KB L2 cache. All the cores on the same socket share a last-level L3 cache of 20 MB. The machine has 128 GB of RAM in total, with half the memory attached to each of the two sockets, and runs 64-bit RHEL Linux 3.10.0.

4.4.1.2 Workload

As discussed in Section 4.2, we used the SmartMart benchmark (Chapter 2) and the REACTDB prototype in-memory actor database system (Chapter 3) for our experiments. We employed the same actor modeling for the application as done previously in Chapter 2. Since our implementation is based on the reactor programming model, we refer to actors as reactors henceforth. Similar to the experiments in Chapter 2, we simulate the workings of one store with eight `Store_Section` reactors. For each, we loaded the `inventory` relation with 10,000 items and the `purchase_history` relation with 300 entries per item for a total of 3,000,000 entries, simulating a history of 120 days where 500 customers on average visit the store per day and buy 50 items each. We fix the number of `Group_Manager` reactors to 10 and vary the number of `Cart` reactors depending on the experiment. The number of `Customer` reactors is set to 30 times the number of carts. To calculate the variable discount, we tuned the window size to correspond to 150 records (see Section 4.4.2) in the purchase_history relation, roughly corresponding to 60 days. The entire size allocated after loading was ~3 GB.

4.4.1.3 System Configuration

For our experiments, we configured the system under the two deployment settings outlined below:

1. sync - In this setting, we employed a single container with an affinity-based router and with eight transaction executors for the corresponding physical cores on the first socket of the machine. Each Cart actor and its associated set of Customer actors were mapped to a disjoint transaction executor, while all the Store_Section and Group_Manager actors were mapped to all the transaction executors. As a result, each invocation of a root transaction on a Cart actor is processed by a unique transaction executor, which also executes all sub-transactions even if to different reactors. We allocated worker threads such that each worker thread generating method invocations on a Cart actor got mapped to the hyper-threaded core of the corresponding cart, simulating client affinity.

2. async - In this setting, we utilized the other socket on the machine to create eight containers with affinity-based routers each consisting of one transaction executor for each of the physical cores on the second socket. Each of the eight Store_Section reactors were mapped to only one container and consequently only one transaction executor. We employed another container similar to the sync setting on the first socket of the machine to map Cart, Customer, and Group_Manager actors. As a result, only a sub-transaction invocation to a Store_Section actor is dispatched to the corresponding remote transaction executor and container for asynchronous execution, while all sub-transaction invocations to other actor types are executed synchronously since they are mapped to the same container. We refrained from employing a mapping similar to Store_Section reactors for the Group_Manager and Customer reactors, because we were limited by the physical parallelism of the machine and we wanted to avoid hyper-threading effects. In addition, the amount of work in the asynchronous method invocation to these reactors is minimal compared to the Store_Section reactor, which would potentially increase execution cost considering the overhead of dispatch as opposed to shared-memory operations.

Since we could only use eight reactors for the Cart actors in the async setting, we only used one socket in the sync setting in order to make comparisons across the two settings. We tuned the thread pool sizes for the transaction executors to minimize queuing delays and maximize usage of physical cores.

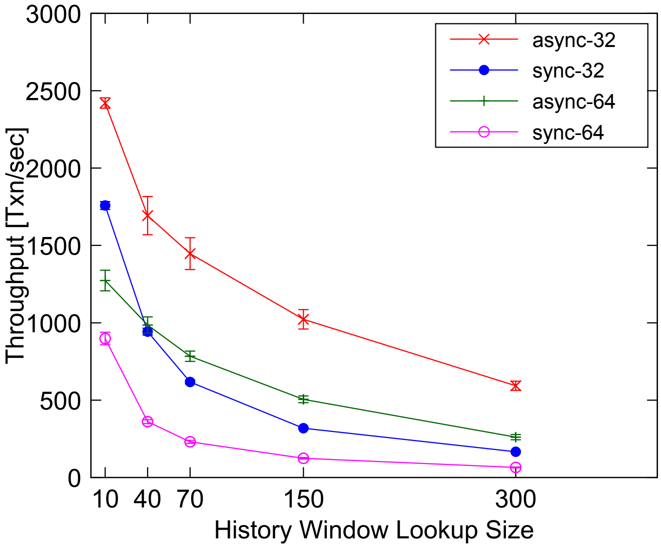


Figure 4.2: Effect of varying history window scan size on throughput.

4.4.1.4 Methodology

As pointed out in Section 4.2, a worker runs interactions consisting of (1) `add_items` and on its successful commit (2) `checkout`. We measure the average latency and throughput of the entire interaction using an epoch-based measurement strategy [51]. We run the experiment for 30 epochs, where each epoch consists of 2 sec. We report averages and standard deviations of successful interactions over the last 20 epochs, ignoring the first 10 epochs that are reserved for warm-up runs. Workers choose customer IDs from a uniform distribution. Unless mentioned otherwise, the items and store sections in orders are also chosen from a uniform distribution for a configurable number of store sections and items per store section in the order.

4.4.2 Effect of Varying Parallelizable Work

In this section, we study the effect of varying the amount of parallelizable work in variable discount computation at checkouts on transaction throughput and latency in the two deployment settings. To avoid interference from inter-transaction parallelism, we run this experiment with a single worker. The number of store sections remains fixed at eight, and the number of items scanned in the `purchase_history` relation for variable discount computation is varied. To avoid skew, the number of items requested from each store section were identical. Finally, to observe the influence of transaction size, we also experiment with two different settings for the number of items requested from each store section, namely either four or eight, thus resulting in a total number of items purchased across all store sections to be

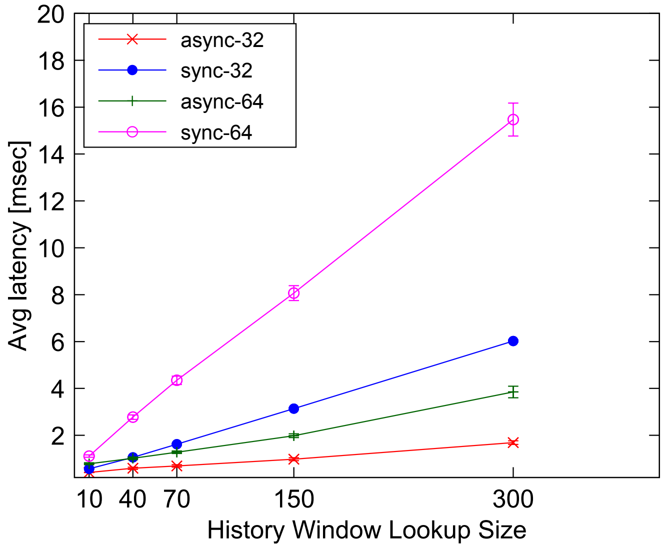


Figure 4.3: Effect of varying history window scan size on latency.

either 32 or 64.

Figures 4.2 and 4.3 depict the throughput and latency results obtained. Since we used a single worker, latency and throughput numbers are almost inverses of each other. We can see that as we increase the history window scan size from 10 to 300, the drop in throughput of sync is higher than that of async. The growing work due to the larger scan sizes and associated mean and standard deviation computations can be parallelized across the store sections in the async version, but not in sync. As a consequence, the curve pairs for the same transaction sizes diverge (this effect can be seen more clearly in Figure 4.3).

Overall, async dominates sync throughout the experiment, due to its ability to exploit intra-transaction parallelism and additional resources even if at the cost of higher overheads. It is interesting to see that as we vary the history window scan size beyond 40, async-64 starts outperforming sync-32, i.e., a sequential transaction is outperformed by an asynchronous transaction of twice the size. Based on the results of this experiment, we choose a history window scan size of 150 for all future experiments, because (1) it is a realistic estimate based on our assumptions of the workload as explained in Section 4.4.1.2, and (2) it allows us to experiment with and demonstrate the effects of intra-transaction parallelism clearly. At that point, the speedup of async-64 over sync-64 is between four and five, which we discuss further in the next section.

4.4.3 Speedup

In this section, we evaluate the classic speedup [49] obtained for the async deployment compared to the sync deployment of the SmartMart benchmark as we vary

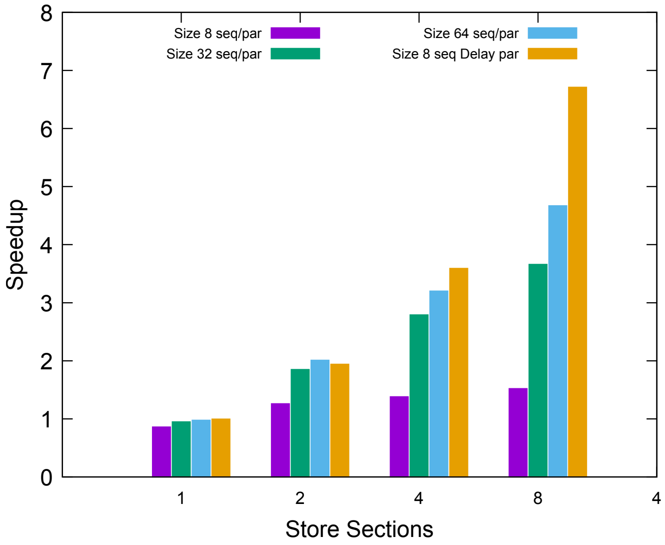


Figure 4.4: Effect of varying parallel work on speedup.

the available parallel resources. In line with our previous discussion, we chose a single worker and a history window scan size of 150 for this experiment. We kept the total number of items in an order fixed and varied the number of store sections that these items are ordered from. For example, when the total number of items is N and the total number of store sections that items are ordered from is k, then N/k items are ordered from each store section. In order to keep our measurements clear, we chose only those values of k for which N is a perfect multiple.

Figure 4.4 shows the throughput speedups for our experiment with three different sizes of N, namely 8, 32, and 64, which are represented by Size N seq/par bars. As an extra control, we created another variant of variable discount computation where we replaced the scan over the history by an artificial delay of 3 msec created by random number generation. This variant models complex calculations that would increase the ratio of parallel to sequential work without increasing the database footprint and the sequential commit cost. The variant is represented by the Size 8 seq/Delay par bar in Figure 4.4. We increased the delay by this specific amount so as to obtain close to 7x speedup, which we pre-calculated based on the ratio of sequential to parallel work in the interactions. Note that we computed the throughput speedup by calculating the ratio of async and sync throughput. The latency speedups in this experiment were almost identical to the throughput speedups and are thus omitted.

We can see that at one store section, we have speedups of slightly less that one. This effect arises due to the lack of asynchronous execution and the small overhead of dispatch. As we increase the number of store sections, however, we can see that the speedups obtained increase as well for all transaction sizes. However, the increase is more pronounced in variants where the ratio of parallel to

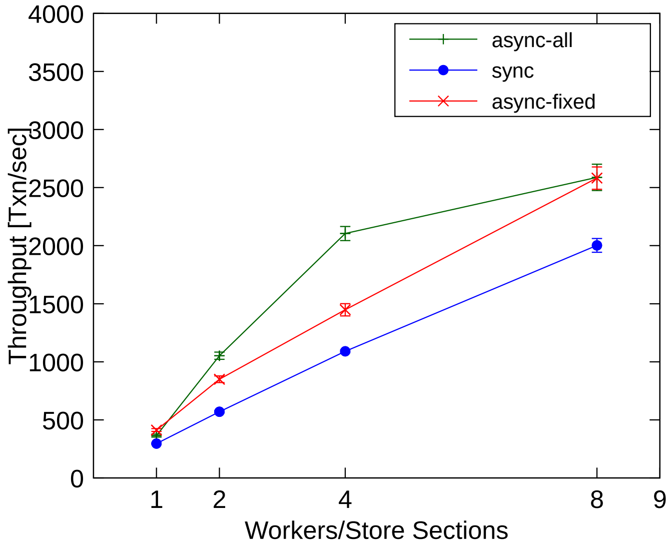


Figure 4.5: Throughput scale-up on both workers and store sections.

sequential work in the transactions is larger. Note that this effect is what is expected by application program structure, since a non-trivial fraction of the complex interaction logic, including `add_items` and parts of `checkout`, is sequential.

At the maximum available parallelism of eight store sections, the speedup of Size 8 seq/par is 1.52, the speedup of Size 32 seq/par is 3.66, the speedup of Size 64 seq/par is 4.67, and the speedup of Size 8 seq/Delay par is 6.8. According to Amdahl's law, in order to get a speedup of 7, 5, 4, 3 and 2 with 8 parallel resources, the parallelizable work must be 98%, 91%, 86%, 76% and 57% of the entire work, respectively. To further drill down into potential sources of overhead, we also profiled the commit costs for Size 8 seq/par and Size 8 seq/Delay par, which were almost the same for both. These costs come to ~68.2 μsec and ~93.8 μsec for the sync and async versions, respectively. For comparison, the latency of the sequential `add_items` function for Size 8 seq/par in sync and async was ~125 μsec, corresponding to one-fourth the entire interaction latency.

In short, even at small transaction sizes, transactional overheads correspond to a small fraction of the latency of the complex application interaction measured. As such, the behavior observed across the different compared variants stems largely from the ratio of sequential to parallel work in the application itself. This results indicates that an actor database system can be used to implement complex application parallelism patterns in transactional code while introducing only small overhead.

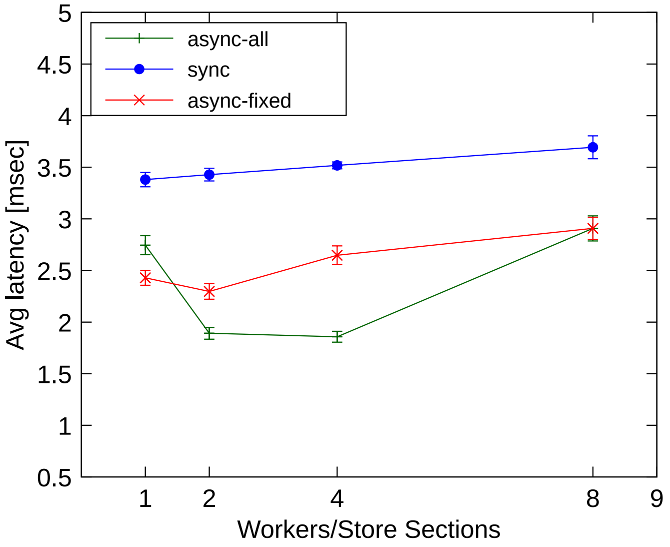


Figure 4.6: Latency in scale-up of both workers and store sections.

4.4.4 Scaleup

The effect of load on asynchronous transaction execution in REACTDB was studied earlier in (Section 2.6.6.3 in Chapter 2). In that experiment, the overall load on the system was varied by increasing the load on each store section as concurrent workers were added, since each worker issues orders across all store sections. As a result, it was observed that async throughput stabilizes when the physical resources allocated to the `Store_Section` reactors reach maximum utilization. In this experiment, we instead increase concurrent workers and physical resources at the same time, thus ensuring that the load on every store section is the same whether we employ one or more workers. This setup corresponds to the classic notion of scaleup [49].

We keep the work fixed to an order size of 32 and vary the store sections that the order spans. For one worker, all the items are ordered from a single store section; for two workers, 16 items are ordered from each store section by each worker across two store sections; for eight workers, four items are ordered from each store section by each worker across eight store sections. Thus, we carefully control the load on each store section, with a total order size of 32 items irrespective of the number of workers. For clarity, we only chose those values of workers of which 32 is a perfect multiple in order to generate identical load across store sections.

Figures 4.5 and 4.6 show the throughput and latency observed. We can see that sync exhibits excellent throughput and latency scalability. This is expected because the sync deployment is agnostic to intra-transaction parallelism and interference effects. We present two variants for `async`. In async-fixed, the store sections for the order are the same across all workers. For async-all, the store sections for the order are chosen uniformly at random from the eight store sections. Thus, async-fixed corresponds to the classical setup for scaleup, while async-all allows extra resources to be leveraged by intra-transaction parallelism when the load on the system is light to normal.

In Figure 4.5, async-fixed also demonstrates excellent scalability and reaches the same peak throughput and latency at eight workers as the async deployment in the experiment in (Section 2.6.6.3 in Chapter 2). By contrast, async-all shows a much higher throughput gain until 4 workers before converging to the same value as that of async-fixed at eight workers. This effect arises because async-all benefits at first from lack of interference across transactions, since store sections are chosen uniformly at random. As the number of store sections in the order increase, the interference effects grow, and async-all converges at eight workers to the throughput of async-fixed. This effect can be clearly seen in the latency curve of async-all in Figure 4.6, which initially shows benefits from intra-transaction parallelism before queuing delays due to increased load cause latency to degrade.

We observe that the performance of async-fixed is higher than that of sync. This is expected because of the asynchronous invocations on store sections and the greater amount of parallel resources available, e.g., one core dedicated to carts and another core to a store section even for one worker. In addition, the benefits are also explained by the lower cost of fixed cross-core communication. Since a transaction spans the same physical cores, this reduces the amount of cross-core traffic. During this experiment, we observed peak abort rates of ~5-7% at eight workers, which is identical to the experiment in (Section 2.6.6.3 in Chapter 2).

4.4.5 Effect of Skew

In this section, we study the effect of skew on the benefits of intra-transaction parallelism. For this experiment, we chose a single worker and kept the work fixed. We use a zipfian distribution to select the store sections that an item is chosen from and vary the zipfian constant to model skew on a store section. Figure 4.7 represent the throughput for an order size of 32 items across eight store sections. We omit a latency plot, since the effects observed match those in the throughput graph. Moreover, we experimented with order sizes of 64 and 8 with delay (as used in the speedup experiments of Section 4.4.3). Since we found the shape of the curves to be identical to the ones presented here, we also omit these results for brevity.

We can see that as we increase the zipfian constant from 0.01 (no skew) to 0.99 (heavily skewed), the throughput of async decreases. This is because the amount of work across store sections is no longer balanced, thus introducing a dependency on the execution cost on the store section with the most orders (increased depth). The throughput of sync is agnostic to skew, since it does not utilize any intra-transaction parallelism. Despite a highly skewed workload when the zipfian constant is 0.99, async still outperforms sync. Even though some store

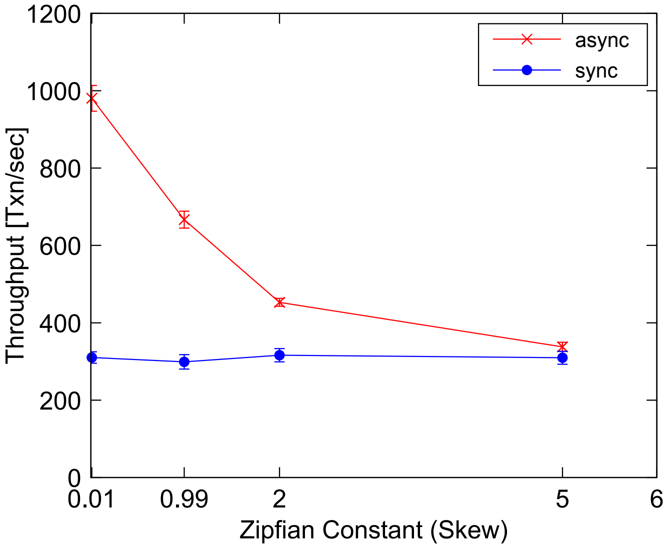


Figure 4.7: Effect of skew on throughput with eight store sections.

sections are more popular than others, all the store sections are yet utilized, leading to intra-transaction parallelism benefits. In other words, while there is a lack of balance across store sections, the result is still better than sequential execution. As we increase the zipfian constant to 5.0, however, the distribution becomes more and more skewed so that all the items in the order are selected from a single store section. At this point, the performance of async converges to that of sync.

4.5 Related Work

Evaluation of transaction processing performance has a rich history, and a comprehensive review exceeds the scope of this chapter. As a consequence, we narrow our focus in this section to contrast our study with the main categories of previous evaluation work of parallelism in transactions.

Many early studies evaluated parallel transaction processing architectures and their impact on concurrency control schemes using performance modeling techniques [5, 29, 36, 37]. Contemporary studies also went further than performance modeling and performed experimental evaluation with nascent parallel database systems [48, 62, 74, 122]. In contrast to our work, all of these simulation and experimentation studies focused on evaluating the performance of disk-based database architectures and their concurrency control mechanisms under synthetic data-access oriented benchmarks, with the exception of [62, 122] that used the TP1 benchmark [59]. Moreover, these early studies only considered intra-transaction parallelism achieved through query speedups by data partitioning [48]. By contrast, our work revisits the effects of the core factors affecting parallelism in the setting of actor database systems with parallelization of arbitrary intra-transaction control flow in a modern benchmark with complex application logic.

Taking note of the popularity of multi-core machines, cloud computing infrastructure and in-memory database systems, recent performance studies have focused on evaluating concurrency control architectures in a distributed infrastructure [67] and on single-node multi-core systems [135]. A separate line of recent work has also focused on evaluating micro-architectural effects of executing OLTP workloads on in-memory database systems [117]. All of these studies focus on evaluating inter-transaction parallelism or high-volume transaction processing and its impact on database architectural components. By contrast, our study focuses on the benefits of intra-transaction parallelism on multi-core machines. In addition to going beyond query-level data parallelism [20, 22, 60, 66, 86], to the best of our knowledge, our study is the first to perform such an evaluation in the context of actor database systems.

4.6 Conclusion

This paper has studied the exploitation by an actor database system of intra-transactional parallelism in a benchmark with complex application logic over multi-core hardware. A few distinctive features differentiate our evaluation. First, all intra-transactional parallelism is exposed to the database system by asynchronous programming constructs in an actor model. This parallelism is not limited to query-level, but extends to both querying and application logic within transactions. Second, we observe that variations in the classic factors affecting parallel efficiency, such as overhead, parallelizable work, interference, and skew, produce the expected effects on intra-transaction parallelization. These results indicate that the evaluated actor database system, ReactDB, introduces only minimal overhead in its implementation of transactional parallelism, suggesting that asynchronicity in actor database systems can be effectively used to bring task-level parallelism gains to emerging latency-sensitive applications with complex application logic. Third, instead of pursuing an exclusive focus on maximum load and transaction throughput, our evaluation illustrates that in a range of light to normal load situations, asynchronicity can lead to reduced transaction latencies by better use of parallel resources. These lower latencies can also engender higher throughputs than classic synchronous transaction execution strategies if idle parallel resources can be put to use by intra-transaction parallelization.

An interesting area of future work is the study of how asynchronicity can be exploited by an actor database system in more complex multi-level parallelization scenarios, including a combination of multi-core and multi-node parallelism and potentially even many-core acceleration. Moreover, we believe that intra-transaction parallelism in complex application logic should be studied more generally by the database community, in contexts encompassing but also going beyond actor database systems.

Chapter 5

Conclusion

> *"Every new beginning comes from some other beginning's end."*
>
> — Seneca

The variety and volume of interactive data intensive applications along with the performance of computing hardware have increased tremendously. Machines with multi-core processors and large-main memory are now commonplace. As a result, there is a widespread demand for scalable high-performance software systems to build and deploy these interactive data intensive applications. Traditionally, database systems have been popularly used to build and deploy these applications because of their transactional guarantees and high-level data model with declarative query support. However, these systems lack primitives for modularity, which makes it difficult to maintain applications as their complexity grows. The lack of a computational primitive makes it hard to reason about the scalability and performance of the application and to flexibly control it.

Of late, actor runtimes have become extremely popular to design scalable interactive data intensive applications because of their actor-oriented distributed programming model and asynchronous message passing semantics. These features provide modularity benefits and allow for reasoning about scalability and performance of the application. However, the lack of state management features in these systems complicates application development.

Our work has been inspired by the observation that these two disparate system domains have complementary strengths and weaknesses. Hence, we proposed the vision of integrating actor features in traditional database systems, thus yielding a new class of systems named actor database systems. In the following sections, we outline the contributions of this dissertation towards the exploration of this vision of actor database systems in order to investigate their potential and viability.

5.1 Summary of the Dissertation

In this dissertation we made three main contributions towards the exploration and development of actor database systems in the context of the current and upcoming interactive data-intensive application and hardware trends:

1. **Actor Database Systems Vision:** The first part of the dissertation identifies a gap in the feature sets of existing systems i.e., actor runtimes and database systems, that impacts interactive data intensive applications. In light of current application and hardware trends, we analyzed how existing systems fail to address these trends. We also highlighted how each of these trends can be addressed by utilizing an actor database system. In order to make the idea of an actor database system precise, we defined the feature set that an actor database system must posses and organized these features them under four broad design goals or tenets. We have postulated the features of an actor database system at a high level intentionally, so as to allow flexibility in the design and implementation of these features. In order to highlight the necessity and applicability of these features, we performed a detailed case study using the SmartMark benchmark, which was designed specifically to represent an emerging interactive data intensive application. In addition to the benefits of modularity, we also demonstrated preliminary performance benefits of the programming model in the context of multi-core machines.

2. **Reactor Programming Model and REACTDB:** In light of the feature set of an actor database system, the second part of the dissertation explores the question of building a scalable high-performance actor database system in the current hardware landscape of multi-core, large main-memory machines. Towards this goal, we introduced a new programming model for relational databases named reactors that provides (1) a computational construct of a logical actor for modularity and reasoning about scalability, (2) asynchronous function invocations across actors while guaranteeing serializable and atomic executions of programs, (3) a cost model to reason about expected program execution costs, (4) relational abstraction of actor state and declarative state query capabilities.

   Instead of building a full-feature actor database system, we first focused on building an in-memory OLTP database engine (REACTDB) exposing the reactor programming model to investigate the challenges that would arise from integrating the programming model inside database engine implementations. We also wanted to validate whether the performance promises of the programming model could be actually realized in practice. In our experiments with classic OLTP benchmarks, we demonstrated that the reactor model can be used to flexibly control the latency of application programs even at the scale of tens of microseconds. Finally, we also demonstrated how the database architecture of REACTDB can be configured across the extremes of shared-nothing and shared-everything at deployment time flexibly without affecting the application programs and their design. We also demonstrated the scalability of REACTDB in our experiments.

3. **Transactional Parallelism in Reactors and REACTDB:** In our experiments with the TPC-C benchmark, we noticed the limited amount of task-level

parallelism available in the application logic, which causes sequential execution of transaction logic to outperform its parallel execution due to communication overheads. In order to understand and quantify the impact of intra-transaction parallelism in the reactor programming model and REACTDB, we performed a thorough investigation of the classical factors that affect parallelism using the SmartMart benchmark, which has higher levels of controllable task-level parallelism in its application logic. Our experiments demonstrated that the available task-level parallelism can be successfully exploited using the reactor programming model in order to better utilize parallel hardware resources. However as expected, the gains from intra-transaction parallelism are diminished as inter-transaction parallelism is increased due to sharing of resources. The experiments demonstrate the use of the programming model to leverage application task level parallelism without worrying about concurrency and atomicity issues, and the efficient low-overhead implementation of REACTDB. The experiments also highlight the flexible deployment configurations of REACTDB, which allows control over parallel vs sequential execution of different program fragments across reactors without necessitating any changes in the application logic.

5.2 Ongoing and Future Work

In this section, we outline several interesting directions of ongoing and future work.

Extending REACTDB for Cloud Infrastructure: REACTDB is currently designed and implemented for high-performance in multi-core machines. Since the architecture of REACTDB (Section 3.3 in Chapter 3) abstracts communication across containers using a transport driver component, current work is underway to extend the system to enable deployment across multiple machines by building a network transport driver component. In such a deployment, containers on the same machine would utilize shared memory for communication while utilizing the network transport driver component across machines. In the design of the network transport driver component, we are exploring the impacts of a fail-stop failure model on the internal architecture of the system. Another interesting challenge that we are facing is to design the network transport driver component efficiently while relying on using the TCP socket API and its guarantees. It remains to be seen how far we can push the scalable deployment of REACTDB across machines before encountering scalability and performance issues with the socket API. Another interesting challenge would be to explore network transport driver implementations using RDMA technologies for performance. Currently, REACTDB employs an OCC protocol on all containers and an atomic commitment protocol across containers if necessary. We anticipate that we would need to revisit our choice of concurrency control protocol for cloud deployments as well in light of current evaluation studies [67]. We also plan to evaluate the impacts of various deployments over cloud infrastructure on system performance.

Multi-Actor Query Functionality: In Chapter 2, we have demonstrated the promise of declarative multi-actor querying in an actor database system by exemplifying this capability in the case study. An interesting challenge is to formalize the semantics of declarative multi-actor querying. As opposed to query support in object-oriented databases [8, 85], multi-actor declarative querying must account for asynchronicity, which can constrain the selection of query plans based on function invocation dependencies in the query. In addition, the declarative nature of query specification must be reconciled with the ability to control latency through asynchronous function invocations.

Deployment Advisors for REACTDB: Currently, in REACTDB the deployment plan, i.e., the mapping of containers to transaction executors and reactors to transaction executors is manually specified. While manual configuration provides a high degree of flexibility and low-level control in tailoring the deployment to a target hardware architecture, it also necessitates specialized knowledge. An interesting avenue of future research is to investigate automatic mechanisms to generate these mappings for a given specification of reactors (based on application design) and application-defined service level agreements. Such an automatic program or deployment advisor should enforce constraints so that the cost model guarantees (Section 3.2.4 in Chapter 3) are not violated, e.g., a reactor should not be mapped to more than one container. Interesting challenges arise on how to deduce the notion of containers from machine characteristics and what criteria should be exposed in the service level guarantees.

Application Defined Durability in REACTDB: REACTDB does not currently support durability. Possible future work in this direction would be to add support for classic transactional durability in REACTDB, by either using classic mechanisms of fast log-based recovery [137] and distributed check-points [55] or by alternatively leveraging guarantees from hardware [52]. The notion of application-defined durability, however, introduces interesting design questions on the implementation of durability mechanisms especially since interactive data intensive applications often have a larger computation footprint than data in their application logic. The impact of application-defined durability on transactional guarantees also needs to be revisited.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[4] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[5] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[6] Akka documentation, May 2017.

[7] Use case of akka, May 2017.

[8] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In *Proc. VLDB*, pages 433–442, 1989.

[9] M. Alomari, M. J. Cahill, A. Fekete, and U. Röhm. The cost of serializability on platforms that use snapshot isolation. In *Proc. ICDE*, pages 576–585, 2008.

[10] G. Alonso. Hardware killed the software star. In *Proc. ICDE*, pages 1–4, 2013.

[11] Amazon go. Store concept website, July 2017.

[12] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS*, pages 483–485, 1967.

[13] J. Armstrong. A history of erlang. In *Proc. ACM SIGPLAN*, pages 1–26, 2007.

[14] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.

[15] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik. The object-oriented database system manifesto. In *DOOD*, pages 223–240, 1989.

[16] M. P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–190, 1987.

[17] P. Bailis. The case for invariant-based concurrency control. In *Proc. CIDR 2015 Online Proceedings*, 2015.

[18] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proc. ACM SIGMOD*, pages 1327–1342, 2015.

[19] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. CIDR 2011 Online Proceedings*, pages 223–234, 2011.

[20] 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.

[21] F. Bancilhon, C. Delobel, and P. C. Kanellakis, editors. *Building an Object-Oriented Database System, The Story of O2*. Morgan Kaufmann, 1992.

[22] C. Barthels, G. Alonso, T. Hoefler, T. Schneider, and I. Müller. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[23] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, 2014.

[24] P. A. Bernstein. Transactional middleware reconsidered. In *Proc. CIDR*, 2013.

[25] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft SQL server for cloud computing. In *Proc. ICDE*, pages 1255–1263, 2011.

[26] P. A. Bernstein, M. Dashti, T. Kiefer, and D. Maier. Indexing in an actor-oriented database. In *Proc. CIDR*, 2017.

[27] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[28] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann, 1996.

[29] A. Bhide and M. Stonebraker. A performance comparison of two architectures for fast transaction processing. In *Proc. ICDE*, pages 536–545, 1988.

[30] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. SOSP*, pages 79–86, 1985.

[31] A. Böhm, J. Dittrich, N. Mukherjee, I. Pandis, and R. Sen. Operational analytics data management systems. *PVLDB*, 9(13):1601–1604, 2016.

[32] J. Briot, R. Guerraoui, and K. Löhr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[33] A. Brook. Low-latency distributed applications in finance. *Commun. ACM*, 58(7):42–50, 2015.

[34] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *ACM SOCC*, 2011.

[35] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. ACM SIGMOD*, pages 413–423, 1988.

[36] M. J. Carey and M. Livny. Distributed concurrency control performance: A study of algorithms, distribution, and replication. In *Proc. VLDB*, pages 13–25, 1988.

[37] M. J. Carey and M. Livny. Parallelism and concurrency control performance in distributed database machines. In *Proc. ACM SIGMOD*, pages 122–133, 1989.

[38] R. Cattell. Scalable SQL and nosql data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[39] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[40] P. K. Chrysanthis, K. Ramamritham, D. W. Stemple, and S. Vinter. The gutenberg operating system kernel. In *Proc. FJCC*, pages 1159–1167, 1986.

[41] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, Sept. 2010.

[42] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proc. ACM SoCC*, pages 163–174, 2010.

[43] S. Das, A. El Abbadi, and D. Agrawal. Elastras: An elastic transactional data store in the cloud. In *Workshop on Hot Topics in Cloud Computing, HotCloud*, 2009.

[44] P. Dasgupta, R. J. LeBlanc, and W. F. Appelbe. The clouds distributed operating system. In *Proc. ICDCS*, pages 2–9, 1988.

[45] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proc. SOSP*, pages 33–48, 2013.

[46] J. Davis, A. Thekumparampil, and D. Lee. Node.fz: Fuzzing the server-side event-driven architecture. In *Proc. ACM EuroSys*, pages 145–160, 2017.

[47] A decade of building facebook, July 2017.

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

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

[50] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proc. ACM SIGMOD*, pages 1243–1254, 2013.

[51] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[52] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. SOSP*, pages 54–70, 2015.

[53] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The bigdawg polystore system. *SIGMOD Record*, 44(2):11–16, 2015.

[54] T. Eldeeb and P. Bernstein. Transactions for distributed actors in the cloud. Technical Report MSR-TR-2016-1001, Microsoft Research, October 2016.

[55] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[56] Erlang documentation, May 2017.

[57] Who is using erlang?, May 2017.

[58] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight multi-key transactions for key-value stores. *CoRR*, abs/1509.07815, 2015.

[59] A. et al, D. Bitton, M. Brown, R. Catell, S. Ceri, T. Chou, D. DeWitt, D. Gawlick, H. Garcia-Molina, B. Good, J. Gray, P. Homan, B. Jolls, T. Lukes, E. Lazowska, J. Nauman, M. Pong, A. Spector, K. Trieber, H. Sammer, O. Serlin, M. Stonebraker, A. Reuter, and P. Weinberger. A measure of transaction processing power. *Datamation*, 31(7):112–118, Apr. 1985.

[60] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *PVLDB*, 8(3):233–244, 2014.

[61] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[62] T. T. P. Group. A benchmark of nonstop SQL on the debit credit transaction (invited paper). In *Proc. ACM SIGMOD*, pages 337–341, 1988.

[63] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. From the internet of things to the web of things: Resource-oriented architecture and best practices. In D. Uckelmann, M. Harrison, and F. Michahelles, editors, *Architecting the Internet of Things.*, pages 97–129. Springer, 2011.

[64] H-store supported benchmarks (smallbank), May 2017.

[65] J. R. Hamilton. Internet scale storage. In *Proc. ACM SIGMOD*, pages 1047–1048, 2011.

[66] W. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. *PVLDB*, 1(1):188–200, 2008.

[67] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *PVLDB*, 10(5):553–564, 2017.

[68] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. ACM SIGMOD*, pages 981–992, 2008.

[69] W. Hasselbring. Microservices for scalability. In *Proc. ACM/SPEC ICPE*, pages 133–134, 2016. Keynote.

[70] Y. Hayduk, A. Sobe, and P. Felber. Dynamic message processing and transactional memory in the actor model. In *Proc. DAIS*, pages 94–107, 2015.

[71] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[72] J. Huang, B. Mozafari, G. Schoenebeck, and T. F. Wenisch. Identifying the major sources of variance in transaction latencies: Towards more predictable databases. *CoRR*, abs/1602.01871, 2016.

[73] J. Huang, B. Mozafari, G. Schoenebeck, and T. F. Wenisch. A top-down approach to achieving performance predictability in database systems. In *Proc. ACM SIGMOD*, pages 745–758, 2017.

[74] J. Huang, J. A. Stankovic, K. Ramamritham, and D. F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proc. VLDB*, pages 35–46, 1991.

[75] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *VLDB J.*, 23(1):1–23, 2014.

[76] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: A scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35, 2009.

[77] G. Kabra, R. Ramamurthy, and S. Sudarshan. Fine grained authorization through predicated grants. *Proc. ICDE*, pages 1174–1183, 4 2007.

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

[79] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. 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. *PVLDB*, 1(2):1496–1499, 2008.

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

[81] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proc. ACM SIGMOD*, pages 579–590, 2010.

[82] P. Larson, editor. *IEEE Data Engineering Bulletin: Special Issue on Main-Memory Database Systems*, volume 36, number 2, 2013.

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

[84] E. D. Lazowska, H. M. Levy, G. T. Almes, M. J. Fischer, R. J. Fowler, and S. C. Vestal. The architecture of the eden system. In *Proc. SOSP*, pages 148–159, 1981.

[85] C. Lécluse and P. Richard. The O2 database programming language. In *Proc. VLDB*, pages 411–422, 1989.

[86] 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 *Proc. ACM SIGMOD*, pages 743–754, 2014.

[87] B. Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.

[88] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of argus. In *Proc. SOSP*, pages 111–122, 1987.

[89] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proc. PLDI*, pages 260–267, 1988.

[90] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. Letter.

[91] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proc. OSDI*, pages 479–494, 2014.

[92] T. Mühlbauer, W. Rödiger, A. Kipf, A. Kemper, and T. Neumann. High-performance main-memory database systems and modern virtualization: Friends or foes? In *Proc. DanaC*, pages 4:1–4:4, 2015.

[93] Mysql customers, July 2017.

[94] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *Proc. CIDR*, 2013.

[95] H. Oakley. Mercury: an operating system for medium-grained parallelism. *Microprocessors and Microsystems - Embedded Hardware Design*, 13(2):97–102, 1989.

[96] Oracle customer successes, July 2017.

[97] Who is using orleans?, May 2017.

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

[99] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: Page latch-free shared-everything oltp. *Proc. VLDB Endow.*, 4(10):610–621, July 2011.

[100] N. W. Paton and O. Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[101] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. ACM SIGMOD*, pages 61–72, 2012.

[102] How paypal scaled to billions of transactions daily using just 8vms, June 2017.

[103] Planet money episode 730: Self checkout. Online podcast, May 2017.

[104] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on hardware islands. *PVLDB*, 5(11):1447–1458, 2012.

[105] Postgresql featured users, July 2017.

[106] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multicore OLTP under high contention. In *Proc. ACM SIGMOD*, pages 1583–1598, 2016.

[107] W. Rjaibi and P. Bird. A multi-purpose implementation of mandatory access control in relational database management systems. *Proc. VLDB*, 30:1010–1020, 8 2004.

[108] L. A. Rowe and M. Stonebraker. The POSTGRES data model. In *Proc. VLDB*, pages 83–96, 1987.

[109] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proc. ACM SIGMOD*, pages 1311–1326, 2015.

[110] Scala documentation, May 2017.

[111] V. Shah. Transactional partitioning: A new abstraction for main-memory databases. In *Proc. VLDB PhD Workshop, Hangzhou, China*, 2014.

[112] V. Shah and M. A. V. Salles. Actor database systems: A manifesto. *CoRR*, abs/17707.06507, 2017.

[113] V. Shah and M. V. Salles. Reactors: A case for predictable, virtualized OLTP actor database systems. *CoRR*, abs/1701.05397, 2017.

[114] R. Shoup and D. Pritchett. The ebay architecture. In *SD Forum*, 2006.

[115] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[116] E. G. Sirer. Nosql meets bitcoin and brings down two exchanges: The story of flexcoin and poloniex, April 2014.

[117] U. Sirin, P. Tözün, D. Porobic, and A. Ailamaki. Micro-architectural analysis of in-memory OLTP. In *Proc. ACM SIGMOD*, pages 387–402, 2016.

[118] The inevitable rise of the stateful web application, June 2017.

[119] Making the case for building scalable stateful services in the modern era, June 2017.

[120] M. Stonebraker. New opportunities for new SQL. *Commun. ACM*, 55(11):10–11, 2012.

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

[122] S. S. Thakkar and M. Sweiger. Performance of an OLTP application on symmetry multiprocessor system. In *Proc. ISCA*, pages 228–238, 1990.

[123] The TPC-C benchmark, May 2017.

[124] Akka transactors, May 2017.

[125] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. SOSP*, pages 18–32, 2013.

[126] P. D. V. Federation at flickr (doing billions of queries per day). In *MySQL Conference*, 2007.

[127] G. Wang, M. A. V. Salles, B. Sowell, X. Wang, T. Cao, A. J. Demers, J. Gehrke, and W. M. White. Behavioral simulations in mapreduce. *PVLDB,* 3(1):952–963, 2010.

[128] T. Wang, R. Johnson, A. Fekete, and I. Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.*, 26(4):537–562, 2017.

[129] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[130] Y. Wen. *Scalability of Dynamic Traffic Assignment*. PhD thesis, Cambridge, MA, USA, 2009. AAI0821759.

[131] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proc. ACM SIGMOD*, pages 31–42, 2007.

[132] W. M. White, C. Koch, N. Gupta, J. Gehrke, and A. J. Demers. Database research opportunities in computer games. *SIGMOD Record*, 36(3):7–13, 2007.

[133] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W. Wong, and M. Zhang. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Trans. Knowl. Data Eng.*, 28(10):2635–2650, 2016.

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

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

[136] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proc. SOSP*, pages 276–291, 2013.

[137] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proc. OSDI*, pages 465–477, 2014.

(The following pages have intentionally been left blank)