Ana Luísa Parreira Nunes Alonso

Database Replication for Enterprise Applications

Universidade do Minho
Escola de Engenharia

The MAP-i Doctoral Programme in Informatics, of the Universities of Minho, Aveiro and Porto

universidade de aveiro

Universidade do Minho

U.PORTO

July 2015

Ana Luísa Parreira Nunes Alonso **Database Replication for Enterprise Applications**

UMinho | 2015

Universidade do Minho
Escola de Engenharia

Ana Luísa Parreira Nunes Alonso

Database Replication for Enterprise Applications

The MAP-i Doctoral Programme in Informatics, of the Universities of Minho, Aveiro and Porto

Universidade do Minho

supervisor:
Prof. José Orlando Pereira

July 2015

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, 28th of July, 2015

Full name: Ana Luísa Pereira Nunes Afonso

Signature: Ana Luísa Pereira Nunes Afonso

Agradecimentos

Ao longo destes anos, tenho tido o privilégio (e o prazer) de trabalhar com um grupo de pessoas excepcional em todos os sentidos: Alfrânio Correia, Bruno Costa, Daniel Machado, Fábio Coelho, Filipe Campos, Francisco Cruz, Francisco Maia, João Paulo, José Marques, Luís Ferreira, Luís Soares, Miguel Borges, Miguel Matos, Nelson Gonçalves, Nuno Carvalho, Nuno Castro, Nuno Lopes, Paulo Jesus, Pedro Gomes, Ricardo Gonçalves e Ricardo Vilaça, uma tendência que continua nos membros mais recentes. Apesar de ir crescendo e mesmo mudando de nome, a essência mantém-se: é um grupo que faz, que acolhe, que orienta e que partilha. E porque a vida também é diversão, fica também o agradecimento a todos os SemEstatuto, onde destaco a influência do Jácome Cunha.

Em particular, quero agradecer ao meu orientador, o Prof. José Orlando Pereira, que embarcou comigo nesta viagem e que mesmo no nevoeiro parece saber sempre que direcção tomar. Se chegámos a bom porto, deve-se a ele.

Quero também agradecer ao Prof. Rui Oliveira, por todo o apoio e também por, em 2007, me ter dado a oportunidade inicial de fazer investigação.

Agradeço também ao Prof. Victor Fonte que, através do GIL, me despertou o interesse pelo trabalho do Grupo de Sistemas Distribuídos.

Finalmente, quero também agradecer aos meus pais, que desde cedo me incutiram o gosto pela ciência e sempre me apoiaram, e àquele que tem sido o meu companheiro nos últimos 11 anos (e meu marido há quase três :)).

Algumas instituições apoiaram o trabalho apresentado nesta tese. O Departamento de Informática da Univesidade do Minho e o HASLab - High Assurance Software Laboratory ofereceram-me as condições necessárias para desenvolver o trabalho conducente a esta tese.

Agradeço também ao Alexandre Sousa que, através da ParadigmaXis, S.A., deu o mote e financiou parcialmente o ínicio do trabalho que deu origem a esta

tese.

O trabalho conducente a esta tese foi parcialmente financiado pelo Sétimo Programa Quadro (FP7) da União Europeia sob o acordo de financiamento número 611068, relativo ao projecto CoherentPaaS.

Braga, Julho de 2015
Ana Nunes Alonso

Intelligence is the ability to adapt to change.

Stephen Hawking

Database Replication for Enterprise Applications

A common pattern for enterprise applications, particularly in small and medium businesses, is the reliance on an integrated traditional relational database system that provides persistence and where the *relational* aspect underlies the core logic of the application. While several solutions are proposed for scaling out such applications, database replication is key if the relational aspect is to be preserved.

However, it is worrisome that because proposed solutions for database replication have been evaluated using simple synthetic benchmarks, their applicability to enterprise applications is not straightforward: the performance of conservative solutions hinges on the ability to conveniently partition applications while optimistic solutions may experience unacceptable abort rates, compromising fairness, particularly considering long-running transactions.

In this thesis, we address these challenges. First, by performing a detailed evaluation of the applicability of database replication protocols based on conservative concurrency control to enterprise applications. Results invalidate the common assumption that real-world databases can be easily partitioned. Then, we tackle the issue of unacceptable abort rates in optimistic solutions by proposing a novel transaction scheduler, AJITTS, which uses an adaptive mechanism that by reaching and maintaining the optimal level of concurrency in the system, minimizes aborts and improves throughput.

Replicação de Base de Dados para Aplicações Empresariais

Um padrão comum no que toca a aplicações empresariais, particularmente em pequenas e médias empresas, é a dependência de um sistema de base dados relacional integrado que garante a persistência dos dados e no qual o aspecto *relacional* é parte integral da lógica da aplicação. Embora várias soluções tenham sido propostas para dotar este tipo de aplicações de escalabilidade horizontal, a replicação de base de dados é *a* solução se o aspecto relacional deve ser preservado.

No entanto, é preocupante que, dado que as soluções existentes para replicação de base de dados têm sido avaliadas utilizando testes de desempenho sintéticos e simples, a aplicabilidade destes a aplicações empresariais não é directa: o desempenho de soluções conservadoras está intimamente ligado à capacidade de particionar a aplicação convenientemente, enquanto que soluções optimistas podem sofrer de taxas de insucesso inaceitáveis o que compromete a equidade das mesmas, em particular no caso de transacções especialmente longas.

Nesta tese, abordamos estes desafios. Primeiro, através de uma avaliação detalhada da aplicabilidade de protocolos de replicação de base de dados baseados em controlo de concorrência conservador a aplicações empresariais. Os resultados obtidos invalidam o pressuposto comum de que bases de dados reais podem ser facilmente particionadas. Assim sendo, abordámos o problema das possíveis taxas de insucesso inaceitáveis em soluções optimistas propondo um novo escalonador de transacções, o AJITTS, que utiliza um mecanismo adaptativo que ao atingir e manter o nível óptimo de concorrência no sistema, minimiza a taxa de insucesso e melhora o desempenho do mesmo.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is a growing awareness of scalability as a key property of enterprise applications. The current target is that IT services are elastic, *i.e.*, that they scale to very large dimensions but also that resources can be provisioned dynamically and incrementally. The motivation for this is twofold: First, to cope with applications with an increasing number of users in a single deployment. Second, to enable the same application to be deployed in increasingly larger settings, allowing a software provider to swiftly capture an emerging market.

At the infrastructure level, this need is being met with the cloud computing paradigm, the combination of a new business model with highly decentralized, scalable, and dependable systems. The infrastructures initially built to meet the internal requirements of large Internet applications such as Google or Amazon.com are currently being commercialized as collections of services that together realize the vision for elastic infrastructure.

Unfortunately, currently available cloud computing proposals fall short in face of the needs of mainstream business applications. Most offerings are for low level infrastructure services like virtual machines and raw storage, which leaves much of the scale issues to the developer. Even initial proposals for multi-tiered application platforms, such as Google App Engine and Windows Azure, are targeted at specific application scenarios and offer limited functionality. Moreover, there are additional dimensions to scalability such as management and maintenance of the application itself, in which an increasingly large number of interventions to keep the system operating and to fulfill changing business needs must be performed.

A particular cause for concern for small and medium businesses is the current

reliance on traditional database management systems, by making use of advanced features or simply by implementing core business logic within the DBMS itself.

In sharp contrast, current proposals for data management in the cloud offer very limited functionality, rely on middleware layers for most processing and lack strong transactional guarantees. Although migrating to a Service-Oriented Architecture is often cited as the long term strategy for scaling, it requires a profound refactoring of current systems, with a large investment in making explicit reliability and performance guarantees that currently are implicit in the usage of the transactional processing engine. These issues create a large gap between current mainstream business applications and the promises of elastic computing.

Database replication differs from object or service replication because of *transactions*. A transaction is a sequence of operations: if the transaction commits, the result of all of its operations are reflected in the database; if the transaction aborts, none of the operations' effects is applied to the database (atomicity, the A in ACID). Also, transactions are required to leave the database in a valid state (Consistency), regardless of concurrently executing transactions (Isolation) and their effects to be persistent (Durability). Even if considering a single database engine, internal threads or processes compete for resources, *e.g.*, read/write access to rows or tables in the context of different transactions. Conflicts occur when two such operations, in the context of different transactions, access the same data item and at least one of them is a write. The purpose of concurrency control in databases, whether replicated or not, is to guarantee that the concurrent execution of transactions over a shared resource, *i.e.*, the database, is correct according to criteria that define how conflicting operations should be handled and to which extent the effects of concurrent transactions are visible to others. Concurrency control is the cornerstone for ensuring isolation.

Concurrency control in distributed systems is harder because processes need to coordinate and agree on which potentially conflicting actions should be performed and in which order, so that the system as whole remains consistent. Database replication requires that replicas somehow agree on which transactions should commit (which can be submitted and/or executed at different replicas) and on which order, so that the replicated database remains consistent.

Thus, the main challenge in database replication is to design efficient concurrency control mechanisms. There are two main approaches: conservative, where

potentially conflicting transactions are detected and prevented from executing concurrently and optimistic, where transactions are allowed to execute concurrently regardless of potential conflicts, which are detected and resolved at commit time.

While, traditionally, relational database systems value (strong) consistency over availability, the need for handling ever-growing sets of data while providing services with high-availability and partition-tolerance sparked the development of key-value data stores, which share at the core a simple key-value data model based on multi-dimensional sorted maps (Chang et al. 2008) with relaxed consistency guarantees. On top of the key-value interface, some offer SQL-like query languages although with restricted functionality (Baker et al. 2011).

Key-value stores (commonly referred to as NoSQL) eschew distributed concurrency control offering restricted (if at all) transactional capabilities. For example, a possible way to avoid distributed concurrency control is to define a primary replica that sequences update operations, while only guaranteeing atomic operations within a single row and performing updates by creating new rows (appending), tagged with either the real-time at which the operation took place or some user-defined time stamp, instead of updating existing ones (Chang et al. 2008). This approach immensely simplifies concurrency control. An improvement to bypass the single-row limitation consists of partitioning data (*e.g.* by user) in a quasi-relational model and mapping each partition to a single row (Baker et al. 2011). Synchronous replication among replicas of a partition in different data centers is handled by a low-latency Paxos implementation (Chandra et al. 2007). Message queues are used to allow replicas to transactionally send multiple messages to replicas of other partitions. However, each recipient processes the message asynchronously in its own transaction, similarly to lazy replication. ACID transactions across partitions require two-phase commit (2PC), which is discouraged in favor of the queues because of increased latency in transactions and higher risk of contention.

In summary, key-value data stores are not suitable for a whole range of applications that either: require SQL, are not easily partitioned, or require frequent, global, fully-ACID transactions with strong consistency.

1.1 Problem Statement

This work was motivated by a production system at a financial operator, which provides brokerage and banking services to both partners and clients. The relational database management system (RDBMS) is the pivotal component of the system, since not only does it provide persistence, but it also serves as the business logic engine: the implicit environment and behaviour guarantees afforded by using a *relational* DBMS should be considered as part of the business logic. This is an architectural pattern frequently employed by businesses and showcases many of the challenges faced by these.

Because this system is business critical, dependability is a core requirement. This means ensuring availability and throughput stability is key: the ability to scale the system as needed will prove instrumental in assuring availability through varying loads, including the occasional load peaks that occur in financial markets.

Replication is often presented as the solution to achieve highly dependable database management services. Existing database replication protocols should be examined to determine to what extent their assumptions and intended behaviour hold in this type of scenario, namely in terms of the need to refactor the application to accommodate them and under heavy load.

The effectiveness of the conservative approach hinges on the characteristics of the workload: (i) the ability to identify such partitions and (ii) the actual number of such partitions that arise. Performance results that have been presented to support such proposals are thus tightly linked to the synthetic benchmarks that have been used. This is worrisome, since these benchmarks have not been conceived for this purpose and the resulting definition of partitions might not be representative of real applications.

On the other hand, replication protocols using optimistic concurrency control fall prey of increasing abort rates when loaded, compromising fairness and throughput. This is also worrisome, since some application domains, such as finance, are particularly sensitive to high rates of aborted transactions.

1.2 Contributions

The first contribution is the **evaluation of the applicability of replication protocols based on conservative concurrency control** using current, more

complex and more realistic benchmark suites, which leads to significantly different conclusions about these protocols' performance and applicability, particularly in the financial brokerage domain. It is important to point out that this contribution is not only directly relevant when considering replication protocols with conservative concurrency control, but it also has a wider applicability to any proposal that assumes that real-world databases can be easily and efficiently partitioned into disjoint partitions.

The second contribution is **AJITTS**, an adaptive transaction scheduler that minimizes aborts, which represent the greatest hurdle for protocols based on optimistic concurrency control. The approach is based on reaching and maintaining the optimal level of queueing in the system, using the adaptive mechanism to introduce finite delays in transaction execution, thus maintaining correctness. Even though this method introduces latency, the net effect is still an improvement in throughput.

1.3 Results

The first result is a **tool** that can be used to analyze SQL-based applications to determine partitioning schemes, focusing on finding disjoint conflict classes.

The second result is the **prototype implementation of AJITTS** in the ESCADA framework. Beyond the actual implementation, because it was done over an abstraction that captures any relational database engine, the ability to do so shows it can implemented anywhere.

1.4 Publications

The contributions presented in this thesis have been partially published in the following papers:

- Improving transaction abort rates without compromising throughput through judicious scheduling.
  Ana Nunes and José Pereira.
  In Proceedings of the 28th Annual ACM Symposium on Applied Computing, 2013.

- Conflict Classes for Replicated Databases: a Case-Study.
  Ana Nunes, Rui Oliveira and José Pereira.
  In Workshop on Planetary-Scale Distributed Systems, 2013.

- Ajitts: Adaptive just-in-time transaction scheduling.
  Ana Nunes, Rui Oliveira and José Pereira.
  In Distributed Applications and Interoperable Systems, 2013.

1.5 Document Structure

This thesis is organized as follows:

- Chapter 2 presents an overview of database replication architectures, focusing on concurrency control mechanisms. In particular, related work on either conservative concurrency control or optimistic concurrency control is discussed focusing on the considered assumptions.

- Chapter 3 presents an evaluation of whether the assumptions made regarding the ability to define convenient conflict classes hold when applied to complex benchmarks or enterprise applications.

- Chapter 4 presents, in detail, the approach behind AJITTS, an adaptive, just-in-time, transaction scheduler, defined over a model that abstracts from implementation details, highlighting its wide applicability.

- Chapter 5 presents a detailed evaluation of AJITTS in a simulated setting.

- Chapter 6 presents a full-featured prototype implementation of AJITTS.

- Chapter 7 concludes the thesis, highlighting contributions and results, and discussing possible directions for future work.

Chapter 2

Background

Database replication has been a hot research topic for some time now, from single-tier architectures to multi-tier and cloud architectures. Currently, distributed transactions are a hot topic with an expanding audience, fostered by STM and cloud databases. Previous work on replication is thus being reused in new settings, widening its significance. The focus has been on how to enable highly available applications/services through fault-tolerant and scalable architectures (Pedone et al. 2003; Kemme and Alonso 2000; Patiño-Martínez et al. 2000; Jiménez-Peris et al. 2002; Kemme et al. 1999). A key concern in the design of fault-tolerant database replication protocols is ensuring that sufficient transactions can be executed concurrently such that the system performs adequately (Correia Jr et al. 2005).

2.1 Overview

2.1.1 Architectures

A naïve approach for distributed coordination is to allow read/write transactions to execute at any node and implement distributed locking: performing a transaction requires synchronous coordination of at least a subset of replicas (with partial replication or all if the data is fully replicated on all replicas).

For example, CacheFusion (Lahiri et al. 2001) (part of Oracle's RAC), which assigns the ownership of each data block to some replica, requires that in performing an update transaction, the ownership of all blocks involved in the transaction

must be transferred to the replica in which the transaction is to execute. This means that in the context of a transaction, while the number of replicas involved in handling a given block of data is limited to three (meta data holder, owner, requesting replica) there is no bound on the number of blocks whose ownership must be transferred, thus severely limiting the scalability of such a system.

Because synchronous distributed locking on a per transaction basis as described above clearly does not scale for large transactional workloads, throughout this work, the focus is on database replication protocols based on other concurrency control mechanisms.

Active replication protocols follow the state-machine approach (Schneider 1990) where the database is considered to behave like a state-machine in which each operation deterministically causes a change in state: each operation is forwarded to every replica, which then executes it. In order for this approach to be applicable, operations must be guaranteed to be deterministic, precluding the usage of current time values and random numbers, as these would likely differ between replicas.

In contrast, in passive replication protocols, commonly referred to as primary-backup, only the primary replica executes the transaction, propagating the transaction's write set to other replicas. The primary's native database engine's concurrency control decides on which transactions to commit or abort and in which order. To insure that replicas remain consistent, these must know or decide on the same serialization order as the primary. In a multi-primary setting, *i.e.*, where different replicas have the role of primary for different parts of data, each transaction still executes in a single primary, but having several primaries means that these must agree on a total order for transaction execution, as a transaction might update data owned by multiple primaries. If replicas apply updates according to that total order, strong consistency is guaranteed.

Group communication protocols that guarantee message delivery with appropriate semantics are instances of the abstract consensus problem (Guerraoui and Schiper 2001) and can be used for that purpose: (Wiesmann et al. 2000) compare different approaches toward replication as well as the primitives needed in each case, while (Défago et al. 2004) present a survey of atomic broadcast algorithms. Figure 2.1 shows how a group communication protocol can be used to order transactions among all replicas: because the total order property guarantees that all

replicas receive the same set of messages and that messages are delivered in the same order to all replicas, the transaction order can be established simply by sending the transaction identifier (along with other relevant meta data) to the group; if transactions are enqueued in the same order in which the respective messages are delivered, the queues at each replica will be identical and can be considered as instances of a replicated queue.

However, waiting for the total order to be established before applying updates introduces a latency penalty. Protocols exploiting optimistic delivery (Kemme et al. 1999) were proposed in an attempt to mitigate the latency penalty, without foregoing strong consistency, at the cost of making replicas implement a slightly more complex concurrency control, that must ensure that the execution based on the order defined by optimistic delivery is correct considering the final total order. It has been shown that this optimization has little impact on the performance of such protocols by challenging the assumption that the bulk of the latency is due to group communication (Correia et al. 2008).

Because active replication requires every replica to execute every transaction, performance is limited by the slowest replica in the group. While passive replication protocols do not suffer from this limitation, transferring large write sets across the network to several replicas can be costly. Protocols that combine active and passive replication have been proposed (Correia et al. 2008). There have also been proposals for mitigating the limitations of state-machine replication, namely by implementing speculative execution and state-partitioning (akin to partial replication) (Marandi et al. 2011) and eschewing non-determinism by restricting valid execution to a single predetermined serial execution (Thomson and Abadi 2010).

Using primary-backup (and multi-primary), ownership of data partitions must be guaranteed to be exclusive. This means that when the primary fails, the database must block until a new primary is found, usually through a leader election protocol. This is costly, particularly in churn-prone environments. Having stand-by fail-over replicas might avoid most runs of the leader election protocol, but at the cost of increasing the number of replicas that need to be updated in each transaction, thereby increasing network utilization and generally increasing the number of nodes in the system without a corresponding improvement in system throughput. Update-everywhere protocols avoid this issue because all

replicas are equivalent. Again, replicas must apply updates according to the defined total order to guarantee correctness.

Database replication protocols can also be classified in terms of when the client is notified the transaction has been committed: in eager (synchronous) replication protocols, the client is only replied to after all replicas have committed the transaction (using, *e.g.*, two-phase commit (2PC)), which can be more costly in terms of latency but provides stronger consistency; lazy (asynchronous) replication protocols reply to the client as soon as the transaction has committed in some replica, later propagating updates to other replicas, providing weaker consistency because of potential temporary divergence between replicas.

An alternative definition is to consider whether updates are propagated to other replicas before the transaction is committed at the primary using a primitive that guarantees delivery and the appropriate message order properties needed by the protocol.

2.1.2 Transactions

A system that meets the *serializability* isolation criterion guarantees that, regardless of the actual interleaving of operations from different transactions during execution, the result is equivalent to some serial execution of the transactions. Serializability, however, does not guarantee equivalence to a particular serial execution of the transactions, but to one of the possible serial executions. Two executions are considered to be equivalent, or more precisely conflict-equivalent, if conflicting operations occur in the same order in both executions.

The overwhelming majority of relational database management systems, however, use *snapshot isolation*, which differs from serializability by considering only write/write conflicts (Lin et al. 2005). Read operations return the latest committed values before the transaction began, *i.e.*, from a snapshot of the database.

Database replication adds a dimension of complexity as other database engines (replicas) are added to the system. These replicas can either host partial or full copies of the "original" database. Partial replication requires partitioning the database either vertically, by assigning sets of tables (or even just sets of table columns) to different partitions, or horizontally, by assigning sets of rows to different partitions, according to some criteria: *e.g.*, key range, key hash, or some composition of criteria.

Correctness criteria for replicated systems can be defined by comparison with the criteria for non-replicated systems, referred to as *1-copy equivalence*. The idea is that the several physical copies behave like a single logical copy, even in the face of failures. An isolation level of *1-copy-serializability* is met when even if failures occur, the execution over the replicated system if equivalent to a serial execution over a single logical copy. Similarly, *1-copy-snapshot-isolation* differs from 1-copy-serializability in that only write/write conflicts are considered.

A way to reason about conflict-equivalence is to consider queues of (potentially) conflicting transactions. Assume that for each transaction t submitted to database D, there is some function $class(t)$ that outputs the set of tables that t might update. Assume for each table T in D there is a queue Q_T that mediates access to T: only the transaction at the head of Q_T is allowed to update table T. Assuming transactions are enqueued atomically across all $class(t)$ queues:

$$\forall t \quad T \in class(t) \iff t \in Q_T \tag{2.1}$$

$$t \underset{Q_T}{<} t' \iff t \quad \text{precedes} \quad t' \quad \text{in} \quad Q_T \tag{2.2}$$

$$Common(t, t') = class(t) \cap class(t') \tag{2.3}$$

$$\forall T \neq T' \in Common(t, t') \implies (t \underset{Q_T}{<} t' \implies t \underset{Q_{T'}}{<} t') \tag{2.4}$$

i.e., for transactions that access sets of tables in common, the relative order in those queues is guaranteed to be the same. In detail: Equation 2.1 states that a transaction t is enqueued in queue Q_T for every table T it accesses; Equation 2.2 defines the partial order relation $\underset{Q_T}{<}$ on the set of transactions as consistent with the precedence relation between transactions in Q_T; Equation 2.3 defines a function that outputs the set of tables accessed by both t and t'; and Equation 2.4 states that the partial orders induced by queues that have transactions in common are consistent.

For a transaction t let

$$Order(t) = \bigcup_{T_i \in class(t)} \underset{Q_{T_i}}{<}$$

be the order relation defined over transactions that is congruent with the set of queues that mediate access to the tables accessed by t. t can only be certified

when it is not preceded by any transaction in $Order(t)$. An equivalent view is to consider a queue congruent with $Order(t)$, denoted as $Queue(t)$: a transaction can only be certified when it is at the head of $Queue(t)$:

$$head(t, Queue(t)) \iff \nexists t' : \quad t' \underset{Queue(t)}{<} t \tag{2.5}$$

Let $H = t_1, t_2, \ldots, t_n$ be a serial history and $D = \{T_1, T_2, \ldots, T_n\}$ a database where:

$$t \underset{H}{<} t' \iff t \quad \text{precedes} \quad t' \quad \text{in} \quad H \tag{2.6}$$

i.e., the precedence relation between transactions in a history H induces a partial order on pairs of transactions, denoted $\underset{H}{<}$. Histories H, H' are conflict equivalent *iff*

$$\bigcup_{T_i \in D} \underset{Q_{T_i}}{<} \subseteq \underset{H}{<} \quad \wedge \quad \bigcup_{T_i \in D} \underset{Q_{T_i}}{<} \subseteq \underset{H'}{<} \tag{2.7}$$

i.e., by definition, serial histories over D are conflict-equivalent if consistent with the partial order defined by the queues over D.

Concurrency control for replicated databases requires replicas to coordinate to ensure correctness. If concurrent transactions conflict, it must be guaranteed that the same outcome is reached in all replicas so that the database remains consistent. *1-copy-serializability* (*1-copy-snapshot-isolation*) is ensured if all replicas have conflict-equivalent serial histories as defined in Equation 2.7.

2.2 Optimistic Concurrency Control

Optimistic concurrency control in distributed data processing systems is increasingly popular. In replicated database systems (Pedone et al. 2003; Kemme and Alonso 2000), it allows concurrent transactions to execute at different sites regardless of possible conflicts. Conflict detection and resolution are performed at commit time in what is known as a certification procedure, before the changes are applied to the database. While optimistic concurrency control allows more concurrency and thus better use of resources than its counterpart, transactions that are later found to conflict must be aborted. In large scale, high throughput transactional systems such as Google's Percolator (Peng and Dabek 2010) and Yahoo's OMID (Gomez Ferro et al. 2014), implementations of optimistic con-

currency control with different isolation levels and locking policies are key to achieving radical scalability.

Certification can either be centralized or replicated. Potentially conflicting transactions must commit in the order in which these appear in the queue. In a centralized implementation, replicas rely on a dedicated participant, *i.e.*, the certifier, to certify all transactions: the certifier maintains a global transaction queue that can be used to determine certification order according to the needs of the protocol. Figure 2.2 shows transactions being submitted to the workers while the global transaction queue is maintained at the certifier, which determines the commit order: assuming that transactions (A to D) are all potentially conflicting, these are ordered such that A should be the first to commit, then B, C and D. Upon completion (*i.e.*, commit or abort), transactions leave the queue.

In a replicated implementation, each participant maintains a replica of a global transaction queue, built using group communication. In particular, group communication is used to totally order transactions, while the replicated queue guarantees that transactions are certified in a conflict-equivalent order. Figure 2.1 shows the replicated queue in three replicas to which four transactions were submitted (A to D), ordered such that, assuming all are potentially conflicting, A should be the first to commit, followed by B, then C and finally D. Considering the conflict-equivalence formalism defined in Section 2.1.2 this can be guaranteed by only allowing a transaction t to be certified if it is at the head of all $class(t)$ queues, or equivalently at the head of $Queue(t)$.

Notice that the more transactions are allowed to execute concurrently, the more likely it is for conflicts to arise. Also, any transaction is vulnerable to being aborted by other transactions from the moment it starts to execute until it is certified: the longer it takes to execute and certify a given transaction, the more vulnerable it is. This is the caveat of most optimistic concurrency control strategies: when loaded, latency increases and fairness is compromised, particularly for long-running transactions, as exemplified with DBSM (Correia et al. 2008).

In an effort to mitigate this issue, several approaches have been proposed. Some use database partitioning to attempt to decrease the load. Others focus on the order in which transactions are executed (scheduling) and/or certified (re-ordering).

One approach is to partition the database defining different certification procedures depending on whether the transaction is local to a single partition or if it reads/writes multiple partitions (Sciascia et al. 2012). While the certification procedure for local transactions relies on a replicated queue, multi-partition transactions require a protocol similar to two-phase commit. Although an optimistic concurrency control mechanism is used, the assumption that the data can be partitioned into disjoint partitions (or that transactions access a very small number of partitions) is the linchpin of the scalability strategy: assuming that the bottleneck lies in the atomic broadcast primitive, it is proposed that partitioning the data can limit the size of the membership for each transaction, breaking up the global membership into smaller groups according to the partitions accessed by each transaction. Considering mostly local transactions, the load on each membership group depends on the number of transactions that access the associated partitions. If most transactions access the same partition, then most of the load is on the associated group, only slightly improving over the global membership scenario. If most transactions are global, then, most of the time, this is equivalent to the original global membership.

Although most optimistic concurrency control protocols execute transactions as soon as these are submitted (Pedone et al. 2003; Kemme and Alonso 2000), it has been pointed out that the worst case scenarios for optimistic concurrency control can be mitigated by limiting the number of transactions executing concurrently (Correia et al. 2008). Transaction scheduling on non-distributed settings using queue-theoretic models for automatically adjusting the maximum parallelism level has been studied (Schroeder et al. 2006a). However, selecting the correct level of parallelism is not straightforward and can result in a severe limitation to maximum throughput.

Galera Cluster is an update-anywhere eager replication solution that uses MySQL-based database engines and implements the replicated queue model. It proposes a mechanism named "flow control" that enables each replica to pause replication in the cluster if the local queue of transactions waiting to be applied grows beyond a given threshold. The threshold is dynamic and grows with the number of replicas in the cluster, as it is expected that more replicas will execute more transactions, therefore requiring greater tolerance in how far replicas can

lag behind[1]. Similarly, replicas that request state transfers cache writes until the transfer is complete and are also able to throttle replication: cluster performance is limited by the apply throughput of these replicas. As proposed, flow control serves only to handle transient spikes in load, performance issues or unusually large transactions: stalling replication can only be done temporarily or queues will grow uncontrollably.

An approach based on state-machine replication with speculative execution would be to execute transactions in batches as soon as optimistically delivered (Hirve et al. 2014). It is assumed that, as long as the sequencer does not crash, the optimistic delivery order will match the final order, ensuring consistency. In short, the idea is to use the latency due to establishing the final order to execute transactions. The number of speculatively executed transactions is statically limited. An evaluation of this approach using the TPC-C benchmark exposes its poor performance with medium to high likelihood of conflict: parallel speculative execution is reduced to serial execution as each speculative transaction has to wait for the preceding transaction to commit in order to execute.

Transaction re-ordering techniques can be used to find alternative serialization orders with the goal of minimizing aborts. One approach is to use optimistic concurrency control with dynamic time stamp ranges (Mahmoud et al. 2014) where the start and commit time stamps of concurrent transactions that potentially conflict are adjusted at each partial replica according to causal constraints: for example, if transaction T reads item x and transactions $T_i, \ldots, T_j$ are concurrent with T and *intend* to write x, then T's and $T_i, \ldots, T_j$'s time stamps must be adjusted so that T's start time stamp is lower that any of the other transactions' commit time stamp, since T has not seen their effects on x. The same type of adjustment is carried out for each item read or written by a transaction and independently at each partial replica, which then cast their vote on whether the transaction can be committed and with which time stamps. While only compared to an implementation of conservative concurrency control using two-phase locking, the results show how vulnerable the approach is an increase in the number of distributed transactions. Also, notice that the overhead of the time stamp adjustment mechanism will likely be significantly more visible with a workload where the probability on conflict between transactions is higher (or with larger

[1] https://www.percona.com/blog/2013/05/02/galera-flow-control-in-percona-xtradb-cluster-for-mysql/

transactions, reading and/or writing more items) as each item will likely have more transactions intending to read or write it.

Transaction re-ordering can also be implemented using a local certification procedure, in a model that is similar to the one in Figure 2.1 (Pedone et al. 1997). In short, if a transaction t cannot be certified in the order in which it was delivered by the total-order group communication protocol, the authors propose analysing the set of transactions that executed concurrently with t, but have already been committed, $C(t)$: for each pair of consecutive transactions in $C(t)$, t_i ordered before t_j, if t's read set does not overlap with the write set of those in $C(t)$ ordered before t_i and t's write set does not overlap with the read sets of those in $C(t)$ ordered after t_j, then t can be certified as if had been delivered between t_i and t_j. Expanding this technique to reordering the whole set of committed transactions is said to lead to an NP-complete problem (Pedone et al. 1997). A refinement would be to allow transactions that have been certified but not yet committed to change their relative order (Pedone et al. 2003). However, because transactions in that state have already acquired write locks, delaying their commits increases lock contention. The number of transactions allowed to be in that state is limited by a constant, determined empirically.

A similar approach to re-ordering, with the goal of preventing read-write conflicts (only considering *1-copy-serializability*) in a partial replication scenario, can be combined with a two-phase transaction execution mechanism that takes advantage of a two-phase commit termination protocol (Diegues and Romano 2013). For transactions where the output of its updates are not used elsewhere in its context, the execution of these operations can be delayed until after all locks have been acquired in the prepare phase of the 2PC termination protocol. These update operations are executed atomically in the commit phase. The assumptions regarding transactions in which update transactions can be safely delayed to the end are similar to those in (Stonebraker et al. 2007) and do not necessarily extend to more complex benchmarks or applications.

Assuming a previously established certification order where t precedes t', it is possible that, if these transactions execute in the same replica and conflict, due to thread interleaving in the database engine, t' may grab its locks before t, blocking t. Because t would have to be certified before t', a deadlock would arise, requiring one of the transactions to be aborted to break it (Correia et al. 2008).

An alternative strategy, would be to swap t and t', if both access the exact same set of conflict classes and assuming t' could be certified. If so, after t' commits, t can proceeed. Otherwise, t' is aborted. One of the issues with this strategy is that this would require knowing which of the potentially conflicting transactions is the one blocking t or to proceed by trial and error. Also, in general, the usefulness of optimizations based on conflict classes is tied to the number of disjoint conflict classes that can be defined for an application.

Another approach is to exploit alternative serialization orders to mitigate the mismatch between the order in which transactions are delivered by optimistic and total-order deliveries (Palmieri et al. 2011).

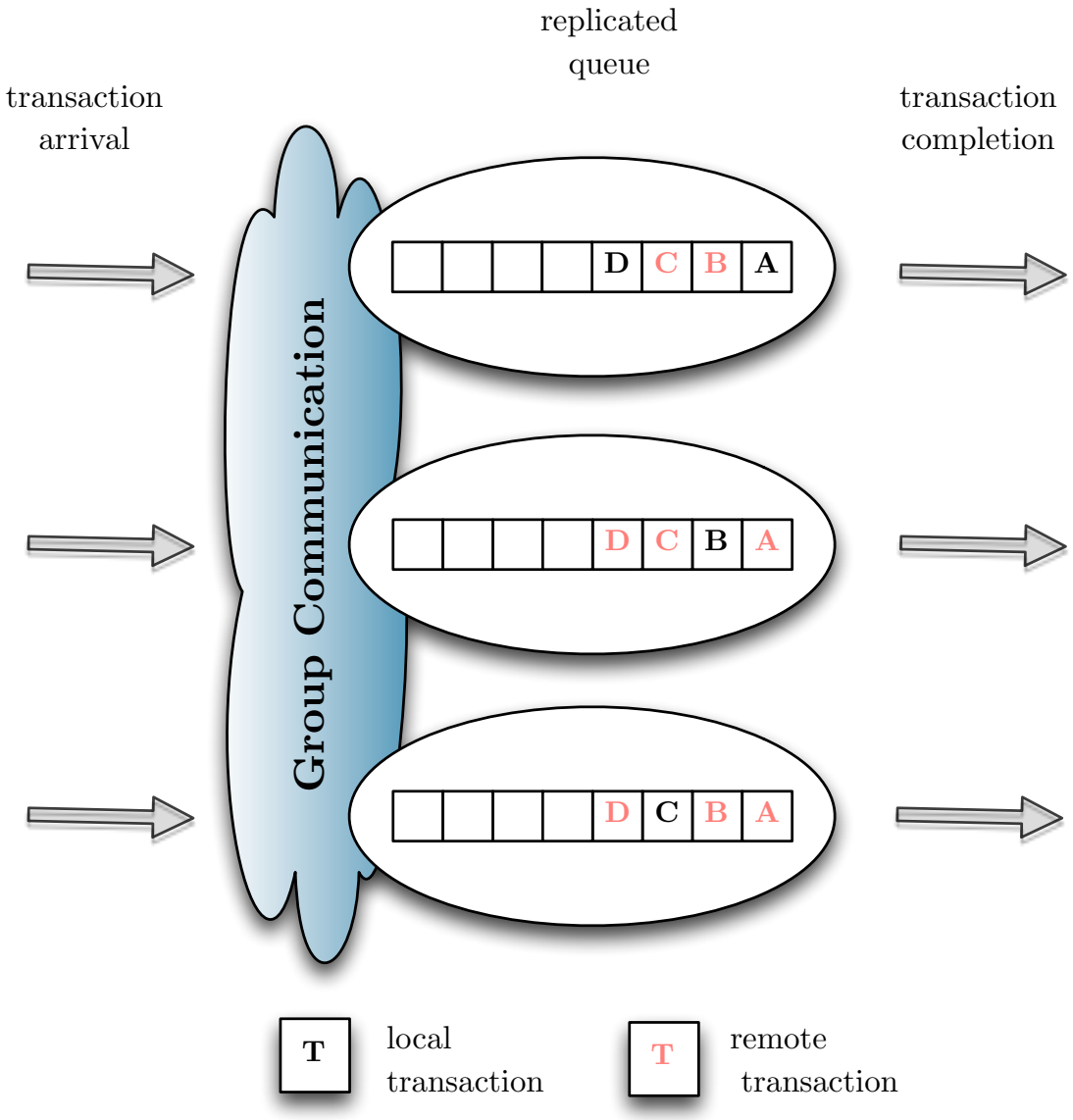


Figure 2.1: Replicated implementation of the global transaction queue.

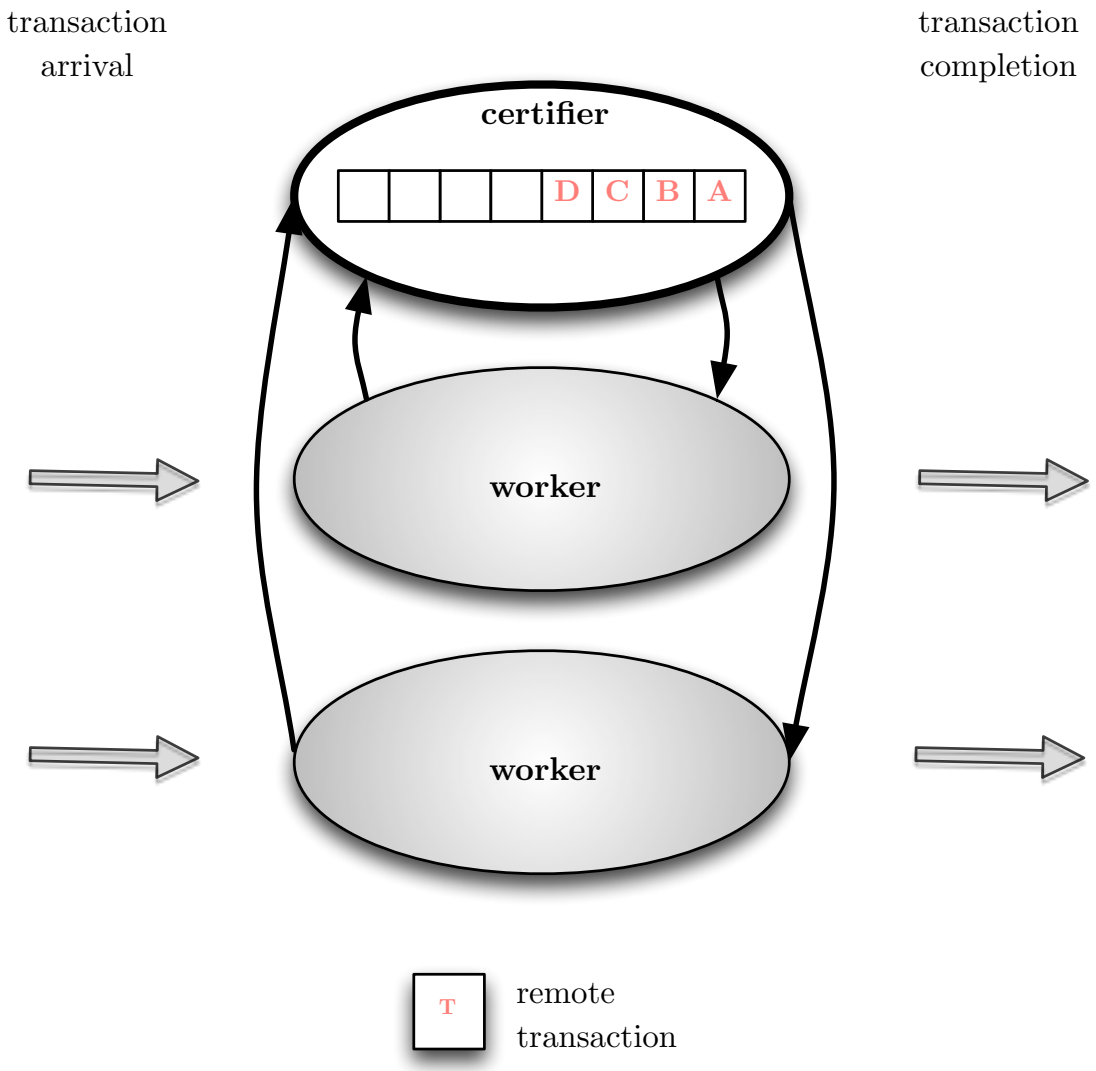


Figure 2.2: Centralized implementation of the global transaction queue.

2.3 Conservative Concurrency Control

A common strategy for assessing potential conflicts is to define conflict classes. In short, the available data is partitioned according to some criteria, and a FIFO transaction queue is associated to each partition (Patiño-Martínez et al. 2000; Jiménez-Peris et al. 2002).

Disjoint data partitions constitute *basic* conflict classes. A straightforward strategy is to map each table to a basic conflict class. Notice that this is a direct implementation of the formalization of conflict-equivalence presented in Section 2.1.2. Compound conflict classes can be defined by grouping basic conflict classes.

Each transaction has an associated set of basic conflict classes according to the data partitions it accesses (as per Equation 2.1). Transactions that access disjoint sets of basic conflict classes are guaranteed not to conflict and thus can be concurrently executed. Conflicts may arise among transactions that access a common conflict class. In order to ensure correctness, those transactions cannot be executed concurrently. This can be straightforwardly guaranteed, using a conservative approach, by only allowing a transaction t to execute if it is at the head of all $class(t)$ queues considering either a replicated or a centralized queue model as defined in Section2.2.

In some conservative approaches, such as the OTP protocol (Kemme et al. 1999), a transaction queue is associated to each basic conflict class. In others, such as the NODO protocol (Patiño-Martínez et al. 2000; Jiménez-Peris et al. 2002), transaction queues are associated to compound conflict classes.

Let conflict classes C_1, C_2 be considered *disjoint* iff

$$\nexists \text{ transaction } t: \quad C_1 \in class(t) \wedge C_2 \in class(t)$$

In any case, the number of transactions allowed to execute concurrently is limited to the number of disjoint conflict classes defined over the database. Thus, the manner in which the database is partitioned is a determinant factor of the performance of a replication protocol using conservative concurrency control. The performance penalty imposed by the conservative strategy depends on the grain considered for concurrency control: if the grain is too fine, conflict detection will result in a delay before transaction execution; on the other hand, if the grain is too coarse, transactions that would not otherwise conflict are needlessly prevented

from executing concurrently. In fact, protocols such as OTP further require that the application can be completely partitioned, since any transaction is restricted to accessing a single basic conflict class.

In order to mitigate this issue, a restricted optimistic policy, guided by which conflict classes are accessed by transactions, can be used for transaction scheduling. Transactions that would abort when executing optimistically are conservatively re-executed, using conflict classes to ensure a conflict-free execution (Correia et al. 2008).

There are also some proposals for transaction scheduling using conservative concurrency control. In (Thomson et al. 2012), the authors propose an architecture for active execution where transaction sequencing and transaction scheduling/concurrency control are handled by different layers. Transactions are ordered by the sequencing layer using synchronous replication via Zookeeper (Hunt et al. 2010), akin to the replicated queue model in Figure 2.1. Each partition's scheduler handles local resource locking for t,ransactions that update the partition: because execution must be deterministic, the order defined by the sequencing layer is strictly followed thus preventing conflicts. After locks are acquired for a given transaction, local reads are performed and sent to every partition that the transaction updates. Finally, the transaction is fully executed but only local writes are applied. A mechanism is proposed to reduce lock contention by minimizing disk stalls: the sequencer delays sending the transaction to the scheduling layer and notifies the storage layer of the data required by the transaction so that when it executes, this data will already be in memory.

2.3.1 Database Partitioning

There have also been some proposals regarding automatic database partitioning, but these either: target data warehousing scenarios, where update transactions are ignored (Rao et al. 2002) or attempt to partition the application in an effort to shift some of the load to an application server (Cheung et al. 2012).

One approach to database partitioning is to optimize for two goals, simultaneously: minimizing distributed transactions, with the restriction of keeping the load on partitions balanced (Curino et al. 2010). This makes it unsuitable for finding partitions accessed by disjoint sets of transactions. Also, the proposed explanation process, an attempt to consolidate classification/query routing rules

to avoid considerably large lookup tables, introduces classification errors: conservative concurrency control based on conflict classes requires knowing exactly which conflict classes will be accessed by a given transaction to guarantee correctness. Another limitation is that statements that access multiple tables are required to use only the attributes used in partitioning rules, or are otherwise not supported. Finally, the result of 12.1% distributed transactions with 2 partitions using the TPC-E benchmark is not sufficient to evaluate the effectiveness of the approach: increasing the number of partitions will almost certainly lead to an increase in the number of distributed transactions. The effect of having less distributed transactions, as the ratio of the number of warehouses over the number of partitions grows, observed using TPC-C, is not likely to happen with TPC-E because the dependencies among possible partitions are not as clear cut.

An improvement is to consider temporal skew when load-balancing, *i.e.*, potential time-related patterns in the workload that create hotspots in particular partitions (Pavlo et al. 2012). Again, because the partitioning algorithm focuses on load balancing it is not suitable for finding disjoint partitions. In particular, while, among others, the TPC-E benchmark is used to evaluate the approach, the effectiveness of the partitioning scheme in terms of transaction throughput for this benchmark is not evaluated.

Another approach is to do fine-grained partitioning instead, *i.e.*, per tuple, using a lookup table (Tatarowicz et al. 2012). However, an analysis of the effectiveness of the approach for TPC-E is done on a restricted subset of TPC-E transactions and tables, thus failing to account for the actual complexity of the benchmark and thus limiting the significance of the results.

A different approach is to do an in-depth analysis of known transaction classes, to determine if these can be transformed to run concurrently, without the possibility of conflicts (Stonebraker et al. 2007). TPC-C transactions are partitioned into sub-transactions that can be executed independently at different sites. In particular, the sub-transactions are found to exhibit the following properties: reads at each site are sufficient to determine locally if the transaction must commit or abort (generally attributed to user data entry errors); there is no communication between sites executing sub-transactions of the same transaction (*e.g.* to communicate intermediate results); sub-transactions are executed conservatively (one at a time) and commutable, *i.e.*, if for any transactions T and T', the sub-

transactions that execute at any site i, T_i and T'_i, produce the same database state regardless of the order in which these are executed if both commit. The ability to convert the TPC-C benchmark to this form exploits the simplicity of TPC-C's schema and small number of transactions and the authors remark that it would be unlikely for an automated partitioning mechanism to reach such a configuration.

TPC-C is too simple to serve as a benchmark to accurately evaluate partitioning tools while partitioning schemes found by automated partition discovery fall short of eliminating distributed transactions: the scheme found in (Curino et al. 2010) while similar to the one defined in (Stonebraker et al. 2007) fails to consider the replication of read-only columns of the *stock* table; even if the discovered partitioning schemes were the same, the actual implementation in (Stonebraker et al. 2007) takes advantage of properties that are not elicited by the tool.

An analysis of TPC-C and TPC-E focusing on the complexity introduced by the latter can be found in (Tözün et al. 2013), namely due to the features that make partitioning significantly more difficult: longer and less deterministic transaction and cross-transaction dependencies.

2.4 Summary

Both conservative and optimistic concurrency control protocols have drawbacks, which have been analysed based on benchmarks that do not mirror the complexity of a real application: schemas that are unrealistically simple are used (*e.g.* TPC-C, or restricted versions of TPC-E) to evaluate database partitioning and consequently conflict avoidance/resolution mechanisms along with the effectiveness of load-balancing or reduction strategies.

In particular, approaches using optimistic concurrency control focus on minimizing the number of aborted transactions, some by limiting the imposed load. Those using conservative concurrency control focus on minimizing contention on transaction execution, relying of the ability to conveniently partition the database to increase performance.

Chapter 3

Determining Conflict Classes

Conservative concurrency control based on conflict classes requires transaction scheduling to adhere strictly to the order defined by the conflict class queues: transactions that access a common conflict class will not be executed concurrently.

In order to use replication protocols with conservative concurrency control efficiently, the data must be partitionable considering the particular data access patterns of the applied workload. Moreover, even if possible, the concrete conflict class definition chosen influences the contention and maximum parallelism attainable. Therefore, the performance of conservative protocols hinges on a favorable definition of conflict classes, as the number of disjoint conflict classes defines the maximum number of transactions that can be executed concurrently.

Most of these protocols have only been tested in particular scenarios with very simple and unrealistic database schemas (Pedone et al. 2003; Kemme and Alonso 2000; Jiménez-Peris et al. 2002; Kemme et al. 1999). Some have also been tested using benchmarks such as TPC-C (Tra 2001a) and TPC-W (Tra 2001b) which have very few database tables as well as few transactions, thereby not reflecting the complexity of analysing a real system to develop a partitioning schema. The question remains whether the assumptions made regarding conflict class definition are still plausible when dealing with more complex benchmarks, for which partitioning is not straightforward at all or, more importantly, regarding real-world applications.

First, The TPC-E (Tra 2010) benchmark, featuring a considerably more complex database model, and a real-world application in the same domain are analysed, focusing on partitioning and the suitability of database replication protocols

with conservative concurrency control for these scenarios. The method used to tackle the complexity of the application in performing this analysis is also described.

3.1 Analysis of the TPC-E Benchmark

The TPC-E (Tra 2010) benchmark simulates the activities of a brokerage firm, which handles customer account management, trade order execution on behalf of customers and the interaction with financial markets.

This benchmark defines 33 tables across four domains: customer, broker, market and dimension. The main transaction types operate across the domains. TPC-E's read/write transactions are: Market Feed (MF), Trade Order (TO), Trade Result (TR), Trade Update (TU) and Data Maintenance (DM).[1]

Unlike TPC-C and TPC-W, TPC-E is an open benchmark suite (Schroeder et al. 2006b): new requests are received by the System Under Test regardless of the completion of previous requests. A closed benchmark suite does not suitably test replication protocols, since the inherent limit to the number of requests received by the system may obfuscate load/contention issues (Correia et al. 2008).

3.1.1 Conflict Class Definition

TPC-E is well-documented and conflict classes can be defined by inspection. In this analysis we considered the 1-copy-snapshot-isolation criterion (Lin et al. 2005): by analysing the database footprint of each transaction type, we determined the specific set of tables for which write/write conflicts between different transaction types can occur.

In order to define conflict classes, the tables read and written by each type of transaction must be inspected to determine which tables are accessed by more than one type of transaction. A naïve conflict class definition is to define a conflict class per table: Table 3.1 depicts the basic conflict classes that can be defined in a table-based manner and the types of transaction that access them. Tables could also be grouped together in a conflict class if accessed by the same set of

[1] The Data Maintenance transaction type operates exclusively on a separate group of tables. As such, it is not relevant for this analysis and is essentially omitted from the discussion that follows.

Table 3.1: Basic conflict classes and transaction types

Conf. C.	Table	Transaction Type
C1	`trade`	MF, TO, TR, TU
C2	`trade_history`	MF, TO, TR
C3	`trade_request`	MF, TO
C4	`cash_transaction`	TR, TU
C5	`settlement`	TR, TU

Table 3.2: Compound conflict classes and transaction types (naïve)

Conflict Class	Transaction Type
{C1, C2, C3}	MF, TO
{C1, C2, C4, C5}	TR
{C1, C4, C5}	TU

Table 3.3: Compound conflict classes and transaction types

Conflict Class	Transaction Type	Write mix
{C1, C2, C3}	MF, TO	48%
{C1, C2, C4, C5}	TR	43%
{C1}	TU_1	3%
{C5}	TU_2	3%
{C4}	TU_3	3%

transaction types, such as tables `cash_transaction` and `settlement`. Table 3.2 depicts the compound conflict classes can be defined based over the basic conflict classes, so that each transaction accesses a single conflict class, as required by NODO. Table 3.1 highlights which transaction types can conflict: because every transaction accesses C1, with conservative concurrency control, all transactions must be serially executed (Patiño Martinez et al. 2005).

TPC-E transactions are composed of frames which makes it possible to define 3 sub-transaction types in lieu of TU. Table 3.3 depicts the compound conflict classes that can be defined considering TU's sub-transaction types TU_1, TU_2 and TU_3. This would allow up to 3 transactions to execute concurrently using a conservative mechanism.

However, after analysing the percentage of transactions of each type compared

Table 3.4: In-depth conflict analysis: (I)nserts, (U)pdates and (D)eletes

	C1	C2	C3	C4	C5
MF	U (PK)	I	D (PK)		
TO	I	I	I		
TR	U (PK)	I		I	I
TU_1	U (PK)				
TU_2					U (PK)
TU_3				U (PK)	

to all write transactions in the benchmark's mix[2], displayed in Table 3.3, we conclude that the majority of the load (91%) is concentrated in two non-disjoint compound conflict classes, {C1, C2, C3} and {C1, C2, C4, C5}, which leads to same performance bottleneck that occurs for the naïve approach. This means that most of the time, transactions will execute serially.

3.1.2 Discussion

Table 3.4 details how each transaction type writes each table. For example, MF transactions update `trade` (C1) by primary key, insert one or more rows in `trade_history` and delete one or more rows from `trade_request` by primary key. Assuming a row-level locking model in the underlying database, concurrent inserts do not conflict, nor do inserts and concurrent updates by primary key, or inserts and concurrent deletes by primary key. Concurrent inserts on `trade_history` (C2) also do not conflict because the primary keys are provided as a part of the transaction's arguments: regardless of the order in which the inserts are executed, the end result is the same. Thus, only MF, TR and TU_1 transaction types conflict. Notice that there is no straightforward way to encode this information in a conflict class definition short of defining one conflict class per row, which would be futile. An alternative might be to explore decision trees, where each node represents a predicate on partitioning attributes and each branch represents the outcome of the predicate, to attempt to consolidate fine-grained partitioning information into broader predicate-based rules (Curino et al. 2010):

[2] We assumed that each sub-transaction is executed a similar number of times, but a different distribution would lead to the same conclusion.

for example, if there is a node with the predicate $x < 5$ and another with $x = 5$ these might be consolidated into a node $x \leq 5$, merging their branches. However, the proposed method for consolidation relies on a heuristic approach which does not guarantee perfect classification of transactions. This is incompatible with correct conservative concurrency control.

The TPC-E recommended way of partitioning the database is to do so by customer identifier, which would effectively partition the trade table horizontally. But, for example, MF transactions update the `trade` table ignoring the customer. Therefore, MF transactions would likely be distributed across partitions. In general, sharding the database would not prove helpful since it would either require: the exact transaction write set to be known before executing it (since conflict detection is done *a priori*); or the transaction to be added to queues of all conflict classes that match shards of accessed tables, thereby rendering sharding useless.

Moreover, current automatic partitioning tools either do not target OLTP systems or produce a small number of partitions, thus being unsuitable for use with conservative concurrency control (as discussed in Chapter 2).

Protocols based on conservative concurrency control offer inherently limited performance in these circumstances (Correia Jr et al. 2005). Still, the TPC-E benchmark can, in fact, be run with a high level of parallelism, *i.e.*, a large number of transactions executing concurrently, with a low abort rate, using optimistic concurrency control as shown on Chapter 4.

3.2 Analysis of a Real-World Application

Our case-study real-world application is a production system at a financial operator which provides brokerage and banking services to partners and clients. The main component of the system is an application that features an architectural pattern frequently employed by businesses, particularly SMEs, and showcases many of the challenges that these face: most features of the brokerage system can be traced to the business logic implemented within a RDBMS, using triggers and stored procedures. Globally, the application consists of hundreds of tables, and thousands of triggers and stored procedures. As a consequence of the development strategy, there is no documentation available, either regarding the

business processes that govern operation, or the interactions and dependencies between them.

The complexity of this application and the lack of documentation exclude the possibility of defining conflict classes by simple inspection, as done in the previous section for TPC-E. A systematic, yet minimally invasive approach is required.

The technique used to determine appropriate conflict classes from the available data is described in detail.

The conflict class definition that results from applying this method to the real case-study application is analysed focusing on the repercussions for conservative concurrency control.

Visual inspection of the application's schema revealed a large number of tables, with many columns, where almost every table was connected through cascading referential integrity constraints, to a large number of tables. There was also a large number of triggers and stored procedures defined in the schema. In short, the complexity of the application meant its analysis would require a more powerful method.

There were some hints of some potential sources of classification:

- tables are grouped into a set of databases;
- there are parameters that are applied to users which will effectively place them in classes, as the flow of an activity will be different according to the value of these attributes, because even if the triggered triggers are the same, the JOINs will select different sets of data to be updated;
- different financial instruments are have distinct execution flows, *e.g.*, derivatives (90% of the business) and non-derivatives; and
- there are essentially two kinds of transactions: deals-related and records-related.

Figure 3.1 shows the number of objects in the database, which hint at the complexity of performing an analysis of the application. These numbers support the need to move beyond simple visual inspection.

The T-SQL source-code was extracted from the RDBMS using the *Generate Scripts* feature of Microsoft's SQL Server Management Studio Express[3].

[3]http://www.microsoft.com/downloads/details.aspx?familyid=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&displaylang=en#Overview

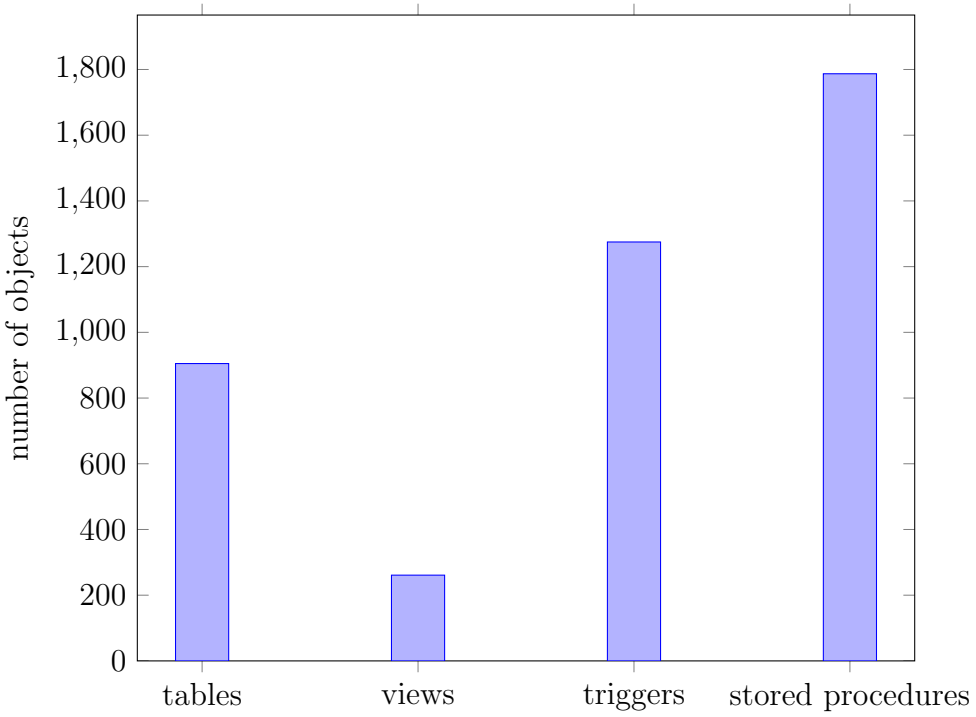


Figure 3.1: Number of database objects per type.

The idea behind this approach was to do a static analysis of the application, by examining the application's source-code, in order to systematically collect information about write operations (INSERT, UPDATE and DELETE statements) as well as about trigger and stored procedure invocations. With this information, the write call graph underlying the mesh of write operations, trigger and stored procedures can be generated. From the write call graph, we can derive statistics and information about the application's structure, which can be used to discover conflict classes as depicted in Figure 3.2.

The necessary information can be extracted from the source code using a parser that selects only information that is relevant to this analysis, making it more robust to syntax variations and more efficient that a generic SQL parser.

Source code analysis

Each database's SQL code is scanned for the following object creation statements:

- `CREATE TABLE`

- `CREATE TRIGGER`
- `CREATE PROCEDURE`
- `CREATE VIEW`

and also for these action statements:

- `EXEC`
- `INSERT`
- `UPDATE`
- `DELETE`

This information is organized into several main data structures:

- for each table, and for each operation (INSERT, UPDATE, DELETE), there is a list of triggers associated to the operation;
- for each trigger (or stored procedure), and for each operation (INSERT, UPDATE, DELETE), there is a list of the tables to be altered;
- for each trigger (or stored procedure), there is a list of the stored procedures called in its body;

Limitations The script does not handle conditional statements. In order to do so, parameter analysis would be required, which would have greatly increased the development complexity. This means that some paths in the graph might not be allowed by the application. Nevertheless, it is unlikely that parameter analysis could be considered when determining which conflict classes are accessed by a given transaction.

Extracting information from log tables

The RDBMS provides an automatic logging mechanism, depicted in Figure 3.3. Only relevant columns are presented in this figure. For each table deemed relevant, each write operation is logged, including the date at which it took place (`TRN_Date`), the user responsible (`TRN_User`) and the machine from which it originated (`TRN_Host_Name`).

In particular, the columns `RecordID` and `TS` seem to be the most interesting:

IDENTITY The `RecordID` column holds a unique (for each table) incremental value, provided by the RDBMS for each row added to the original table.

TIMESTAMP The `TS` column holds a time stamp which is simply the value of a counter, provided per database, that is incremented on each insert or update operation on any table with a time stamp column, in that database. The time stamp value associated to a given row refers to the latest operation.

After placing operations within transaction borders, we can reason about the transaction itself, at a higher level of abstraction. Using the information stored in these columns, operations executed within a database can be ordered. Across database boundaries, however, no order can be accurately established. The ability to order operations within a transaction makes the information extracted more accurate and therefore easier to compare with the graph. Log data was collected by a one-time query on the production database encompassing a time period considered to be representative of the application. The information stored in the log tables is used to prune graph paths that are not used in the application.

Limitations The tuple consisting of `TRN_Date`, `TRN_User` and `TRN_Host_Name` columns was initially considered as the key for mapping operations to transactions. However, this information is not sufficient to establish this mapping. Also, the exact type of the write operation cannot be determined just by examining the log tables. This limitation simply makes the results a little less accurate.

Graph Generation

The parser generates a directed, single-edged call graph, which can be directly converted to the *dot* language format, part of the Graphviz project [4].

There is a vertex in the graph for each write operation applied to a given table. There is also a vertex for each trigger or stored procedure. An edge (a, b) implies that method (or operation) a calls or triggers method (or operation) b.

In cases where a vertex a has children b and c, this means that both b and c are called by a. The order in which b and c executed cannot be determined from the graph. Notice that for any given vertex, its successors are executed within the same transaction.

[4] `http://www.graphviz.org`

Possible executions For any given vertex, the set of all of its successors matches the set of operations that might be executed atomically with the operation represented in that vertex. For a given vertex a consider the subgraph, induced by the write call graph on the vertex set consisting of all of a's successors. Consider the tree in Figure 3.5 as such a subgraph. In this case, as determined by a depth-first traversal starting at a, a possible execution would be:

- $a, b, c, d, e, f, g, h, c, i, j$

but, for example,

- $a, d, e, f, g, h, c, i, j, b, c$
- $a, d, e, f, i, g, h, c, j, b, c$

would also be possible, because no particular execution order among a vertex's successors can be assumed.

Depth-first traversal mimics the nesting behaviour of calls.

Figure 3.4 shows a subgraph of the application's global graph, using vertex *INSERT:dbB:tbl_A* as the origin. This subgraph shows which vertices (and edges) can follow an INSERT on table *tbl_A*.

The format of the vertices' labels conforms to one of the following:

- $< (INSERT|UPDATE|DELETE) >:< database\_name >:< table\_name >$
- $< database\_name >:< trigger\_name > \_ < (insert|update|delete) >$
- $< database\_name >:< procedure\_name >$

In terms of the nomenclature exhibited by the application, the following generally applies:

- A stored procedure vertex can be recognized by the *_sp_* in its name.
- The *SEC_* prefix identifies validation triggers, which are always executed first.
- The *LOG_* prefix identifies triggers that perform logging, which always execute last.

For example, an INSERT on table *tbl_A* of the *dbB* database causes the trigger *dbB:proc_tbl_A_insert* to be fired which may insert rows on table *tbl_I* and/or on table *tbl_D*. For most tables, there are validation and/or logging triggers that respectively precede and/or follow INSERT, UPDATE and DELETE operations. These are ommitted from Figure 3.4 for ease of presentation.

Notice that while the subgraph contains cycles, this does not mean that the execution flow will also contain cycles. For example, an INSERT on table *tbl_A* *may* trigger an update on table *tbl_J*, which *may* call procedure *dbB:proc_tbl_J_vcc* which *may* insert some rows on table *tbl_A*.

As a side note, if a write call graph were generated for a stored-procedure-based implementation of, *e.g.* TPC-E, the stored procedures that implement the main logic of each transaction type would appear in the graph as vertices without predecessors.

3.2.1 Conflict Class Extraction

Figure 3.6 shows an example of a write call graph for a complete application, featuring table write operations, triggers and stored procedures: an INSERT on table A triggers Trigger_1A to do an INSERT on table B and calls stored procedure Stored_Proc_1; an UPDATE on table B triggers Trigger_2B to do an UPDATE on table C thereby calling Stored_Proc_2; Stored_Proc_3 updates table D but it is only called by interactive users. From this graph, it can be deduced that in the application that originated it, no transaction ever writes table B and table D as there is no path in the graph connecting a write operation on B to one on D. It also means that transactions that write on table B might also write on table C. Table 3.5 shows a conflict class definition considering each table as a basic conflict class: for example, conflict class $C1$ corresponds to table A and is accessed by an unknown set of transaction types represented as S_1. From the graph,

$$(S_1 \cup S_2 \cup S_3) \cap S_4 = \emptyset$$

resulting in the disjoint compound conflict class definition in Table 3.6. The goal is to derive disjoint compound conflict classes automatically from the graph.

Table 3.5: Basic conflict classes and sets of transaction types.

Conf. C.	Table	Transaction Type
C1	A	S_1
C2	B	S_2
C3	C	S_3
C4	D	S_4

Table 3.6: Compound conflict classes and sets of transaction types.

Conflict Class	Transaction Type
{C1, C2, C3}	$S_1 \cup S_2 \cup S_3$
{C4}	S_4

By aggregating INSERT, UPDATE and DELETE operations on the same table in a single vertex, we get a simplified graph, which offers a more data-focused view of the application as depicted in Figure 3.7. The aggregation allows information from the log tables to be incorporated into the graph: each vertex can be annotated with the number of writes for the corresponding table found in the log tables.

A directed graph (or a directed component) is weakly-connected *iff* in the undirected version of the graph, for each pair of vertices, there is a path between them (Chartrand and Lesniak 1996). In terms of this specific analysis, each weakly-connected component corresponds to a self-contained set of triggers, stored procedures and INSERT, UPDATE or DELETE operations that can be executed within the same transaction. Notice that for any given transaction, the set of tables it writes is contained in a single weakly-connected component. Assume there are N components and that T_i is the set of tables for which there is a write vertex in component i. The following properties hold for weakly-connected components:

$$\forall i \neq j \in \{1..N\} \quad TS_i \cap TS_j = \emptyset \tag{3.1}$$

and

$$\forall i \in \{1..N\} \quad \bigcup_{i=1}^{N} TS_i = \Omega \tag{3.2}$$

where Ω is the set of all tables. Assume that a conflict class C_i is defined as the set of tables in TS_i. From Equation 3.1 we can conclude that this method results in disjoint conflict classes, one per component. From Equation 3.2 we can conclude that every read/write table is considered in a conflict class.

Figure 3.8 shows the weakly-connected components that result from the graph in Figure 3.7, which matches the definition presented in Table 3.6.

Improving component detection

After analysing the graph of the example vertex's component in Figure 3.4, it is clear that the *dbB:sp__debug* vertex is serving as a "hub" connecting otherwise possibly unrelated components. By removing this vertex from the graph, which would represent a hypothetical application refactoring where this stored procedure would be replaced by others, the number of weakly-connected components increases significantly. Consequently, the number of extraneous vertices within each component is reduced. For example, the example vertex's component shrunk from 3500 to 2480 vertices. However, in order to accurately evaluate the data from the log tables together with the graph, hub vertices should be considered. As long as these hubs are sink vertices, this is not an issue. It is possible that other such vertices exist, and if appropriate, their removal from the graph can make the latter more manageable with more finely-tuned components. Sink vertices with large in-degrees are good candidates for this type of analysis.

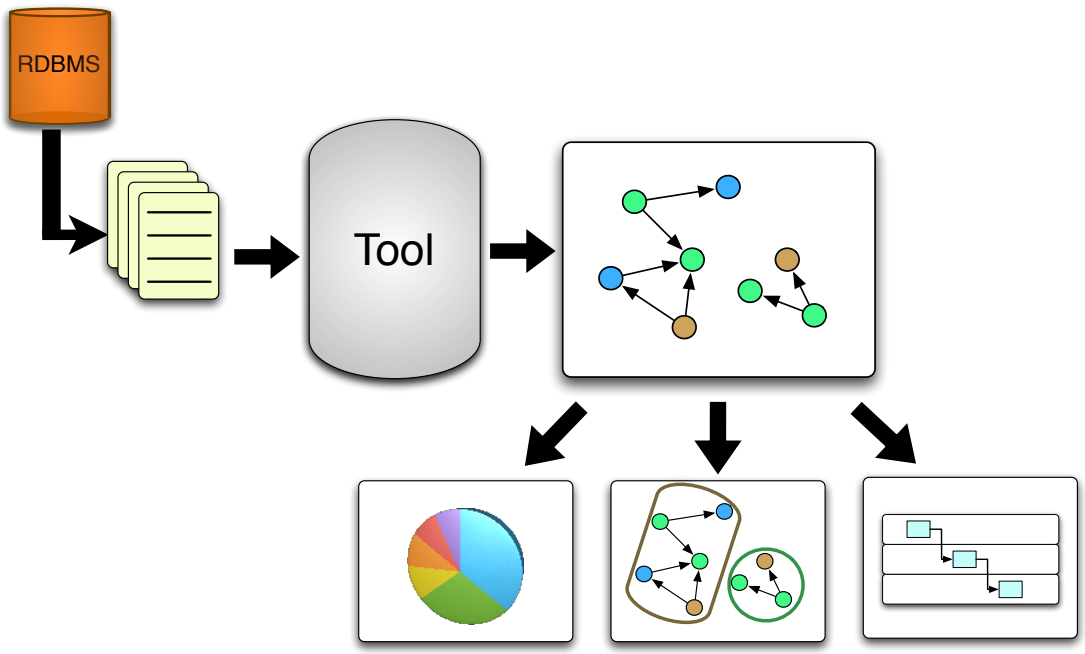


Figure 3.2: The source code is extracted from the RDBMS and analysed by the tool, which then generates a write call graph, from which statistics and information about the application's structure can be derived.

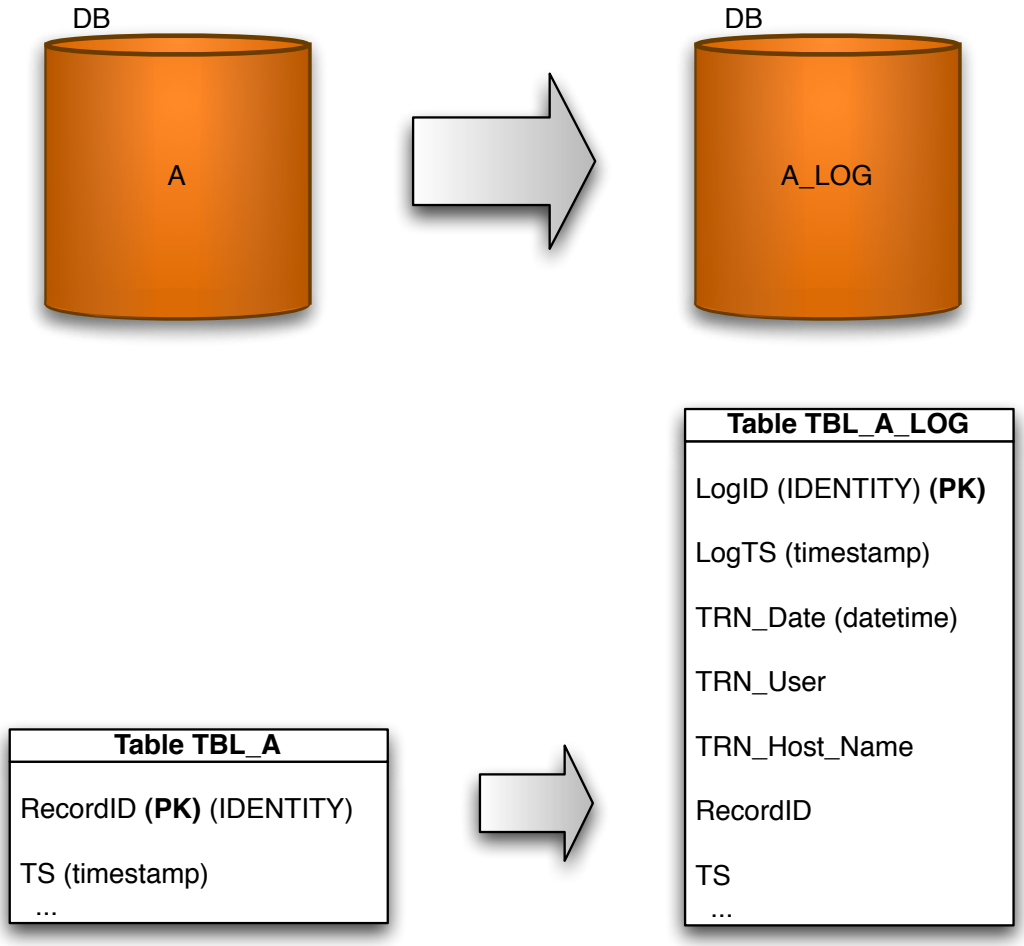


Figure 3.3: The RDBMS creates a log database where logs are stored. For each table in the original database, there is a corresponding table in the log database, to which the `LogID`, `LogTD`, `TRN_Date`, `TRN_User` and `TRN_Host_name` columns are added.

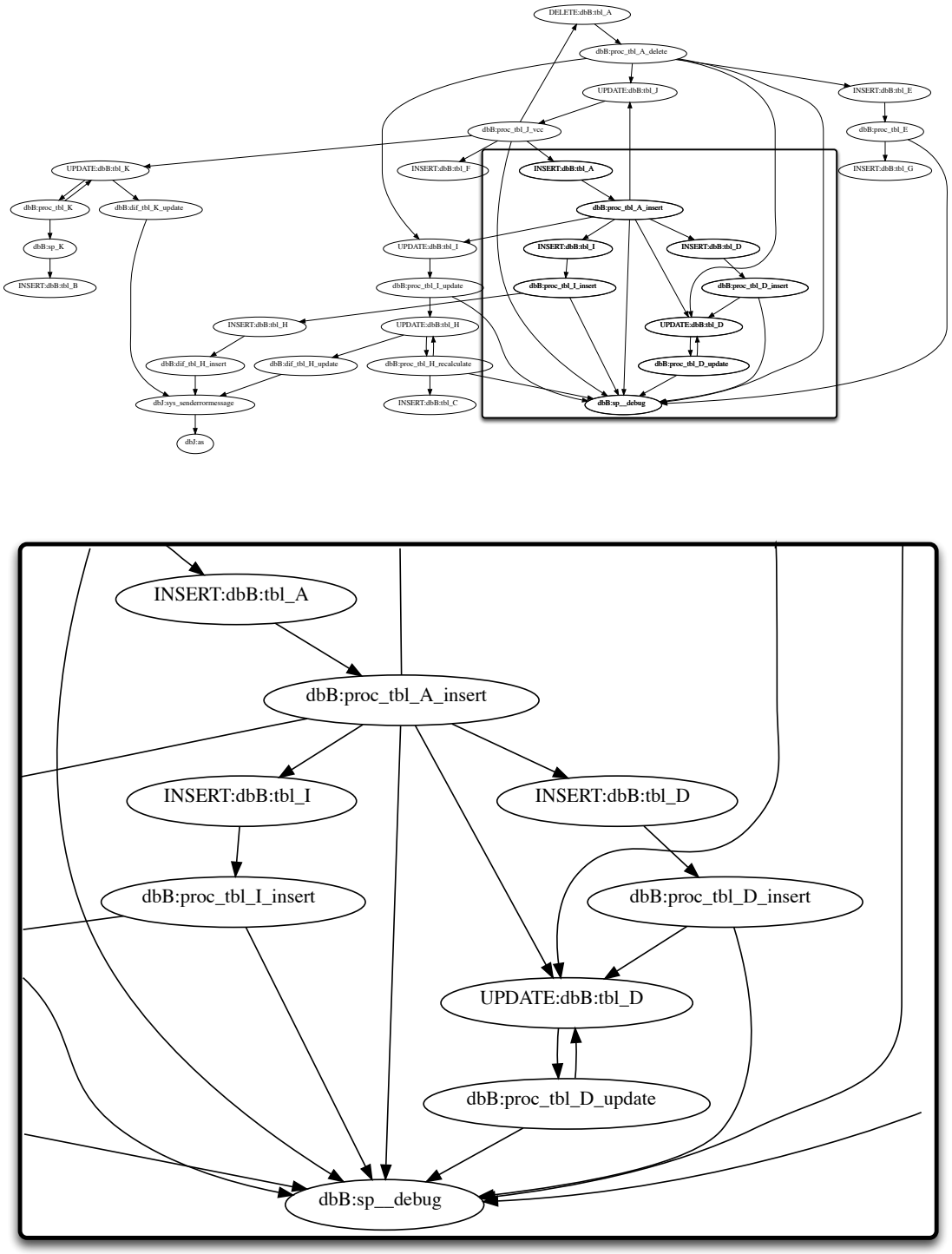


Figure 3.4: An example graph: vertices correspond to operations (INSERT, UPDATE, DELETE), triggers and stored procedures. Edges correspond to calls. Above, the subgraph using *INSERT:dbB:tbl_A* as the origin. On the bottom, a portion of the subgraph, in greater detail.

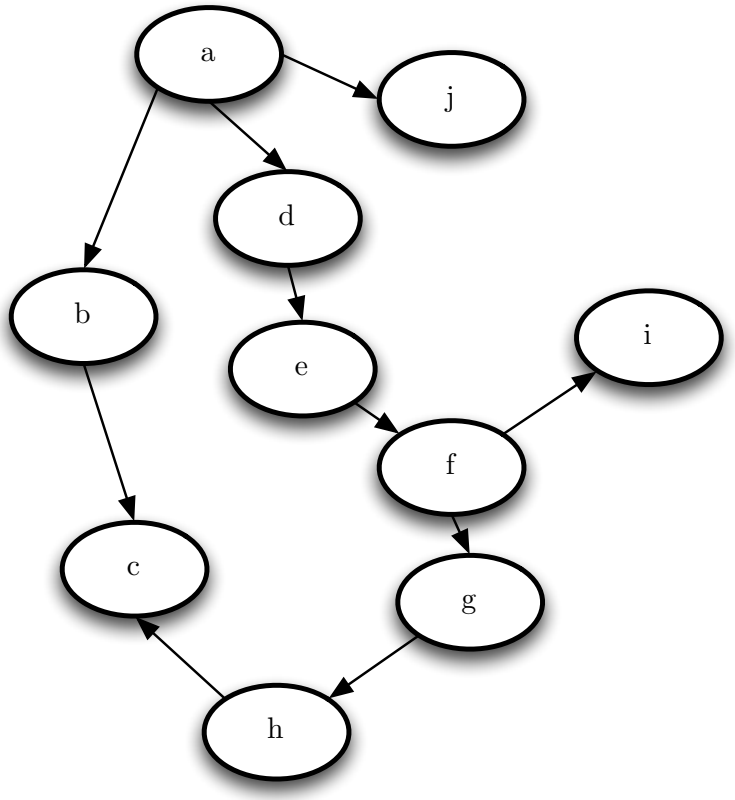


Figure 3.5: An example subgraph.

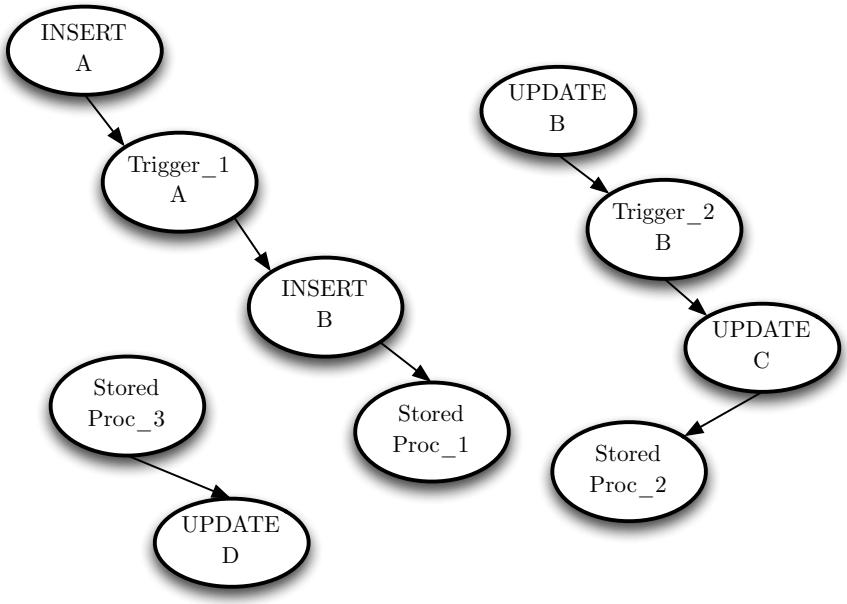


Figure 3.6: An example of a write call graph.

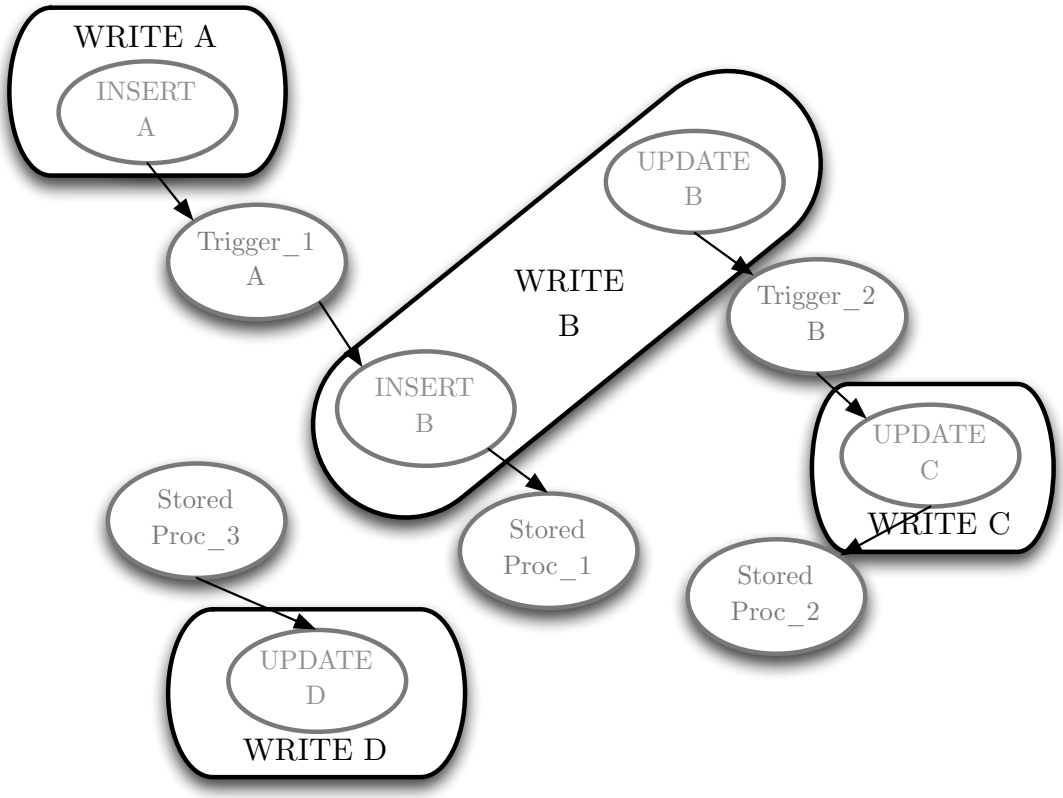


Figure 3.7: Vertices representing write operations on the same table are aggregated into a single vertex, which simplifies the graph. Other vertices are folded into the edges, further highlighting the connections between tables.

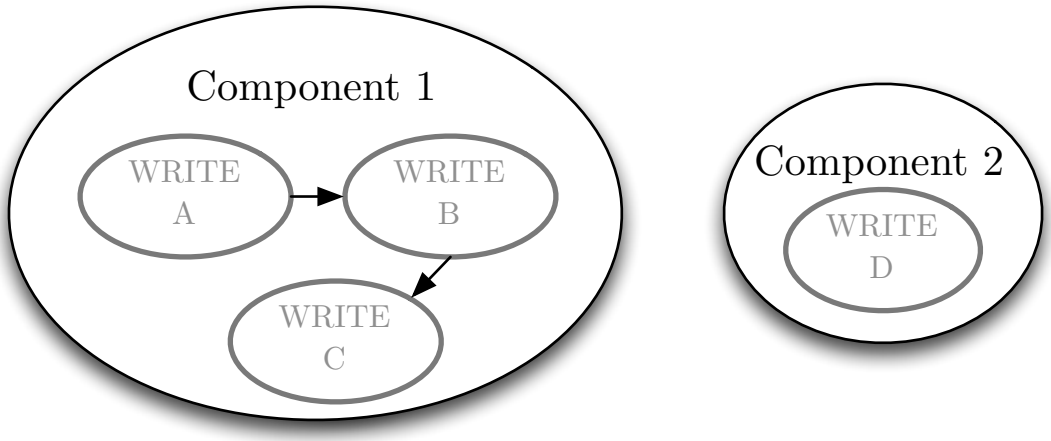


Figure 3.8: Weakly-connected components that result from Figure 3.7.

3.2.2 Discussion

The application is structured as several databases in the same RDBMS. Aggregating vertices that operate on tables of the same database and collapsing multiple edges between databases as a single edge decorated with the number of collpased edges, shows that except for databases *dbK* (which is a remote database, not considered to be a part of the application) and *dbA*, the databases are significantly entangled, (Figure 3.9)). Database boundaries are of practically no avail for defining disjoint conflict classes.

Applying the extraction method to the case-study application resulted in the identification of 130 weakly-connected components, which, as stated, correspond to the same number of disjoint conflict classes.

For this application, considering any replication protocol with conservative concurrency control based on conflict classes, at most 130 transactions can be scheduled to execute concurrently. This seems to be a very promising result.

However, upon examining database logs, we found that most transactions access the same conflict class (component/partition). Therefore, almost no transactions can be executed concurrently, leading to higher contention that originally expected.

The obvious way to circumvent this issue is to partition the troublesome component. In an effort to do so, the component with the largest number of writes was analysed in search of *cut vertices*: vertices that, if removed from a graph, result in an increase of the number of components (Chartrand and Lesniak 1996).

Of the 90 cut vertices found, the one that would lead to the largest number of new components (8) was selected. For each new component, Table 3.7 shows its size and the number of transactions that write on it.

Note that partitioning the graph like this would require that the table that corresponds to the cut vertex could be added to each of the 8 new components,

Table 3.7: New components, their size and number of writes.

Component	c1	c2	c3	c4	c5	c6	c7	c8
Size	1	3	28	12	135	2	10	4
Writes	0	50268	5172	4033	394738	359	4	474

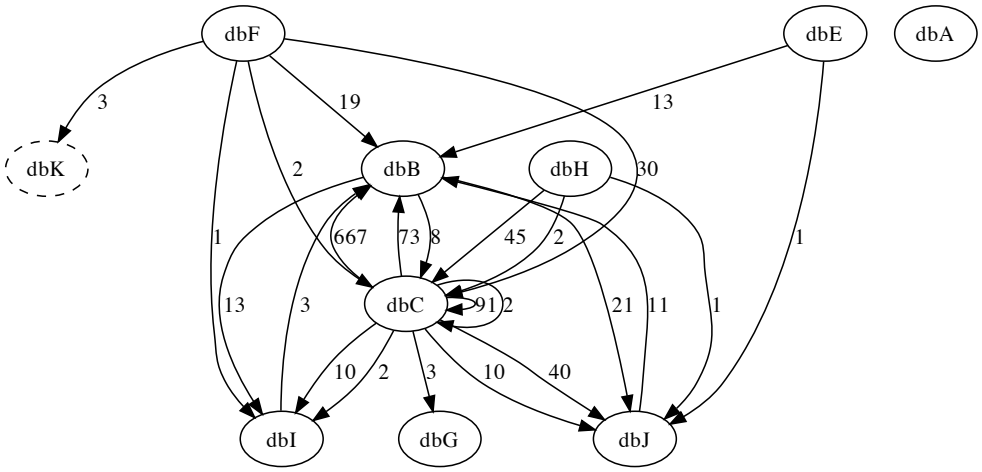


Figure 3.9: Nodes correspond to databases and edges are labeled with the number of operations that cross database boundaries.

making these new partitions correspond to non-disjoint compound conflict classes. While some protocols with conservative concurrency control such as NODO are based on non-disjoint compound conflict classes, increasing the number of non-disjoint classes does not increase the level of concurrency allowed by the protocol.

An alternative would be to partition the table that corresponds to the cut-vertex, creating 8 new disjoint conflict classes. This would, however, require refactoring the application and re-structuring the database. Even assuming that this could be done, the vast majority of the writes remain concentrated in a single component (c5). The cut-vertex strategy could now be used to partition component c5 and so on. Still, the bulk of the writes targets a single table. The next step would be to partition the heavily-written table, which would necessarily lead to table partitioning based on filters over its attributes. In this case, matching accessed items to conflict classes would amount to a satisfiability problem, particularly considering that all "cut-vertex" tables must also be partitioned (Correia Jr et al. 2005; Guo et al. 1996). In short, the same issues that made sharding unsuitable in Section 3.1.2 are also applicable to this scenario.

Although this technique is simple, it is exaustive (we tried removing all nodes and selected the ones that yield the most partitions) and optimistic (we are not sure that these partitions could actually be realized by refactoring) thus providing

a very strong counter-argument. We can safely conclude that no easy refactoring exists such that effective conflict classes based on syntatic criteria can be defined.

Partitioning this application is much more complex than partitioning a TPC-E database. Moreover, while the number of disjoint conflict classes that can be defined for this application is much larger than what can be reasonably defined for TPC-E, it does not result in a pratical advantage when considering a conservative concurrency control mechanism. This scenario presents a significant hurdle for the performance of replication protocols with conservative concurrency control, which are thus unsuitable.

Notice that such a scenario would not, however, emcumber a replication protocol with optimistic concurrency control: even if most transactions write on a common table, but on different rows, no conflicts occur.

3.3 Summary

Replication is often used to achieve highly dependable database management services, however, if the result is unable to cope with the actual workloads it can be self-defeating, as the service grinds to a halt with peak loads. The extent to which assumptions of existing protocols hold in the real world have been examined: there was not, to the best of our knowledge, published work that provides a concrete counter-example that can be cited. This is precisely what makes it significant.

First, the TPC-E benchmark was examined and despite being well-structured in terms of schema and transactions, the number of disjoint conflict classes that can be reasonably defined was found to be very small, which implied that protocols based on conservative concurrency control are not suitable for this type of application.

Then, a real-world brokerage application was analysed. To enable this analysis a general method was devised for extracting a partitioning scheme based on a graph derived from the application's source code. In this case, the number of disjoint conflict classes that could be defined was significantly higher than in TPC-E. However, after looking at the distribution of write operations per tables, the vast majority of the write load were found to fall on a single partition, and at greater detail, on a single table. While this table could be horizontally

partitioned, it is not clear how to do so to benefit a conservative concurrency control mechanism. It is shown that even if the application could hypothetically be refactored leading to a larger number of conflict classes, most transactions would still conflict.

Again, the performance of a protocol based on optimistic concurrency control would not be hindered by these facts and one might speculate it would achieve high concurrency in this setting. The next step is to evaluate the behaviour of replication protocols based on optimistic concurrency control for this domain of application. But, notice that these conclusions also mean that optimizations based on conflict class information for protocols based on optimistic concurrency control will tend to have a minimal effect.

Chapter 4

Scheduling Optimistic Execution

If a transaction must wait to be certified in the correct order to ensure consistency in a distributed system, it is better that it waits prior to execution, when it is not susceptible to being aborted by conflicts with concurrent transactions. The implementation of this simple intuition does however imply that the system is continually monitored and that an appropriate execution start time is found for each transaction. We propose AJITTS, an adaptive just-in-time transaction scheduler. First, a system model that captures the most relevant aspects of distributed transaction processing systems is described. Then, AJITTS is introduced.

4.1 System model

We assume that transactions are submitted by clients to a distributed database system where they can be executed optimistically. A certification procedure ensures that no conflicting transactions are committed. Transactions are totally ordered and thus, currently submitted but yet undecided transactions can be regarded as a queue. In this queue, transactions are executed and go through a sequence of states several states until leaving the system being either committed or aborted (Figure 4.1). It is also assumed that aborted transactions are not automatically re-executed, as the decision to re-submit them is left to the client.

This is described by the abstract Algorithm 1. In detail, upon submission, transactions are enqueued (line 3), assigned the *not executed* state (line 4) and transition to the *executing* state once execution begins (lines 5 and 6), transition-

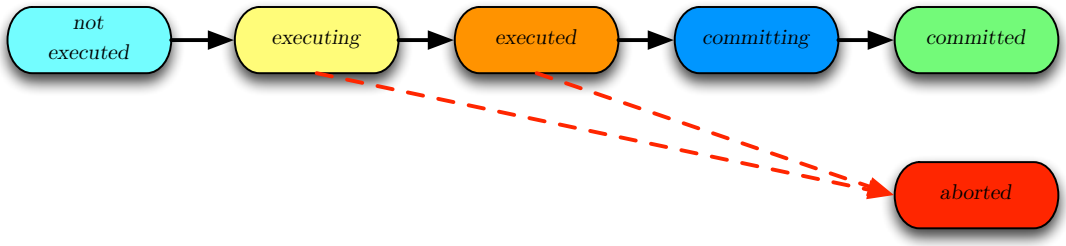


Figure 4.1: Allowed transitions between transaction states.

ing to the *executed* state as execution ends (line 10). A transaction remains in the *executed* state until it is its turn to be certified according to an order that is conflict-equivalent to the previously agreed commit order, verified by the predicate $head(t, Q)$, *i.e.*, until there is no potentially conflicting transaction in Q that precedes it. (line 12). Notice that while a transaction may satisfy $head(t, Q)$ in any state, it will only be certified after it has finished executing without having been aborted, *i.e.*, in the *executed* state (line 12). If the transaction can be certified, it enters the *committing* state (line 14) and the database is notified of the decision to commit the transaction with high-priority (line (15), progressing to *committed*, when complete (lines 20-22). High-priority is needed to ensure the committing transaction can acquire all necessary locks, regardless of being held by transaction that have not been certified. If the transaction cannot be certified, the database is notified that must abort (lines 17), progressing to the *aborted* state when done (lines 24, 25). When complete, transactions are removed from the queue (lines 22, 26).

Notice that,

$$Queue(t) = \{t' \in Q : Common(t, t') \neq \emptyset\} \implies Queue(t) \subseteq Q$$

so from Equation 2.5, $head(t, Q)$ can be defined as

$$head(t, Q) \iff \nexists t' \in Q : \quad t' \underset{Q}{<} t \wedge Common(t, t') \neq \emptyset$$

Because parallel certification requires that the database can be partitioned into disjoint conflict classes (or some equivalent abstraction) and we show in the previous chapter that enterprise applications (as well as complex benchmarks) do not fit this assumption, we will consider all transactions as potentially conflicting,

so that a transaction satisfies $head(t, Q)$ when it is at the head of Q, since Q becomes equivalent to $Queue(t)$.

Algorithm 1: Abstract base protocol algorithm.

```
1  t: transaction, Q: queue, not_executed, executing, executed, committing,
   committed, aborted : states ;
2  upon t is submitted
3  |  enqueue(Q,t);
4  |  t.state ← not_executed;
5  |  execute t;
6  |  t.state ← executing;
7  |  return;
8  end
9  upon t is executed
10 |  t.state ← executed;
11 end
12 upon (t.state == executed ∧ head(t,Q)
13 |  if certified(t) then
14 |  |  t.state ← committing;
15 |  |  commit t;
16 |  else
17 |  |  abort t;
18 |  end
19 end
20 upon (t is committed)
21 |  t.state ← committed;
22 |  dequeue(Q,t);
23 end
24 upon t is aborted
25 |  t.state ← aborted;
26 |  dequeue(Q,t);
27 end
```

Because transactions must be certified in a conflict-equivalent order to the total order on which replicas agreed, the system can be modelled as a single queue, to which all transactions are submitted. This models either a centralized ordering at a transaction manager server (Gomez Ferro et al. 2014) or a distributed ordering built using a group communication system (Pedone et al. 2003) as depicted in Figure 4.2.

Figure 4.2 shows how the centralized and replicated certification models (re-

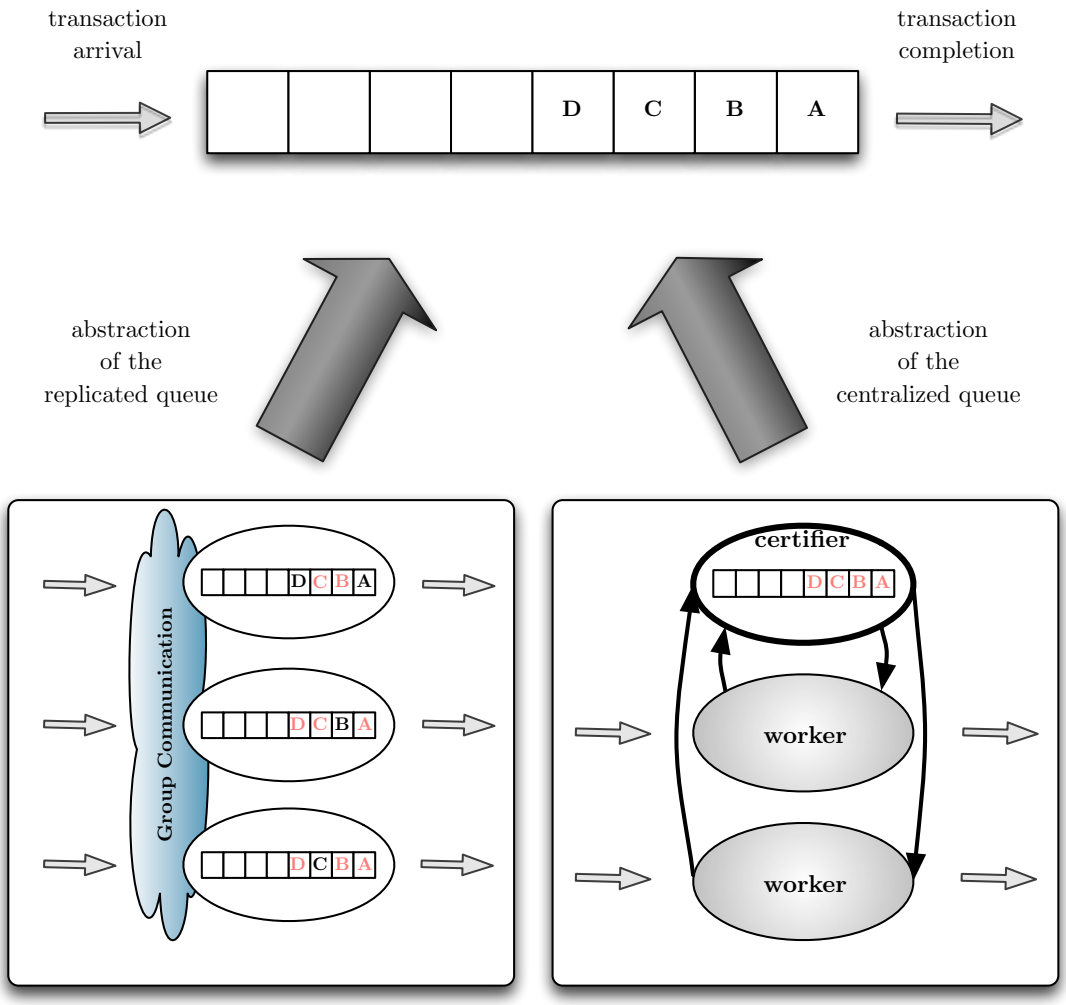


Figure 4.2: System model as an abstraction of both a distributed and a centralized queue.

spectively in the bottom right and bottom left frames), already described, can be represented in the system model: transactions (A to D) ordered such that, assuming all can potentially conflict, A must be certified first, followed by B, C and then D are submitted to the queue from the left; transactions are considered to be local, abstracting away communication delays; upon satisfying $head(t, Q)$, transactions are certified, leaving the queue upon being committed or aborted.

The certification procedure present in Algorithm 1 abstracts from the implementation details and models either: (a) implicit (or in-core) certification (Kemme and Alonso 2000) where transactions that satisfy $head(t, Q)$ in the *executed* state (line 12) are inherently certified, so $certified(t)$ always returns *true*; or

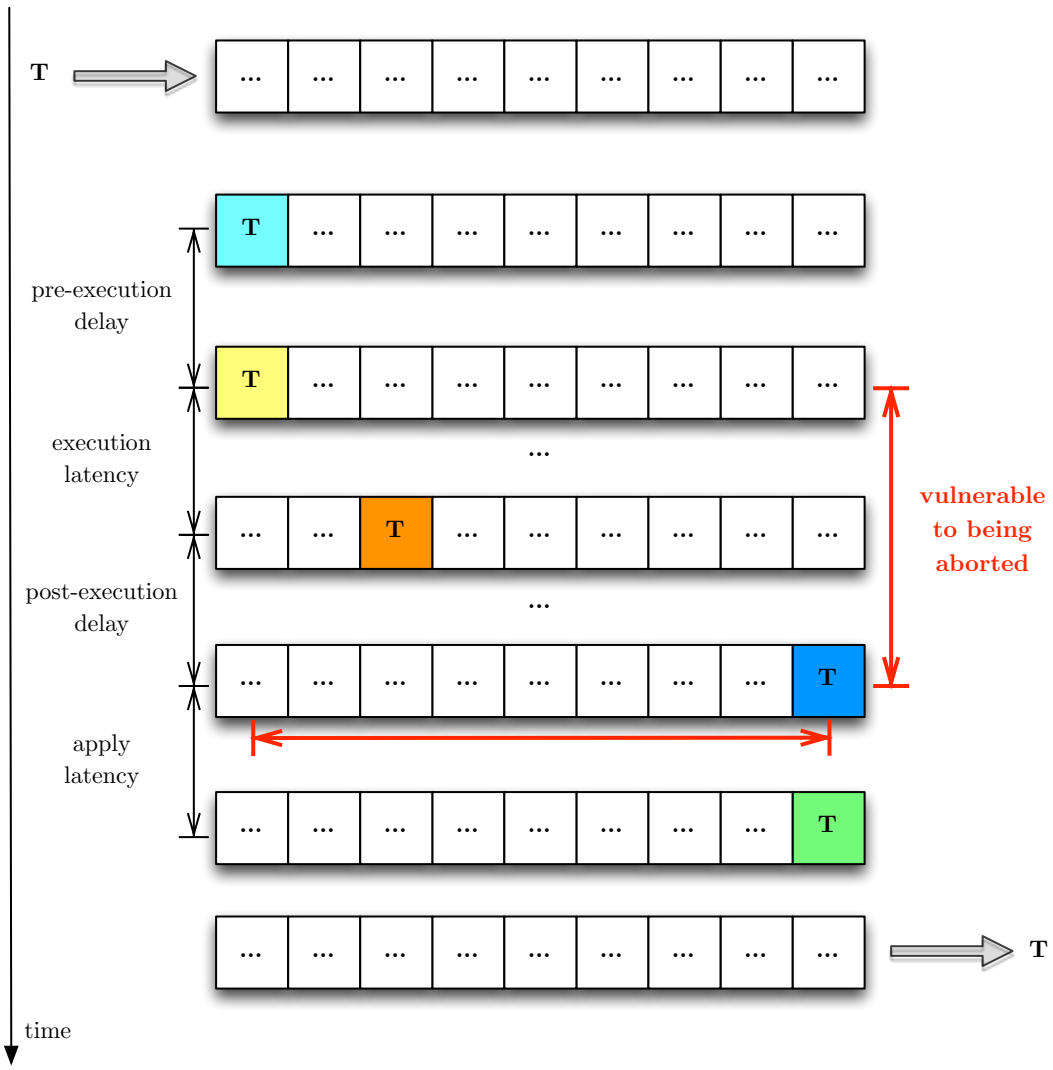


Figure 4.3: Transaction life cycle events and the time intervals these define.

(b) explicit certification, where *certified*(t) returns *true* if the t does not conflict with concurrent transactions that have already been committed. Notice that in either case, committing a transaction t causes conflicting transactions in either *executing* or *executed* states to be aborted.

Snapshot isolation is assumed, which differs from serializability by considering only write/write conflicts (Lin et al. 2005) (line 24). This is used in the overwhelming majority of current RDBMSs and has also been favoured in distributed transaction processing systems.

Transactions are vulnerable to being aborted by committing transactions since their execution starts until reaching *committing*, *i.e.*, during *executing* or *executed*

states. Figure 4.3 depicts this as a vulnerability window: because transactions are executed immediately upon submission, these are vulnerable throughout their term in the queue. The longer transactions are vulnerable for and the higher the number of transactions executing concurrently, the more likely it is for aborts to occur.

4.2 Approach

The main insight leading to our proposal is as follows: as transactions are vulnerable to being aborted from the time execution starts until being certified, in order to minimize the number of aborts, execution should start as late as possible. On the other hand, if there are no transactions ready to be committed because the transactions that should be certified still have not completed execution, throughput decreases. Our approach is thus based on reaching and maintaining the optimal schedule for starting transactions: as late as possible to minimize aborts but as early as needed to maximize throughput.

Instead of executing transactions immediately upon submission, the number of transactions executing concurrently can be throttled by placing a threshold in the queue: transactions *below the threshold* are not considered as eligible to start executing, while transactions beyond the threshold that are in the *not_executed* state are to be executed (lines 29-33) of Algorithm 2. Simply put, transactions are evaluated for eligibility to execute whenever a transaction arrives to (*i.e.* is submitted) (line 5) or leaves the queue (*i.e.* committed (line 22) or aborted (line 27)).

The first image of the queue in Figure 4.4 shows transactions A, B, C and D, so ordered, all in the *not_executed* state and a threshold placed so that there can be at most 2 transactions ahead of it. Only A and B are *beyond the threshold* and thus eligible to be executed as shown in the second image of the queue. As A finishes executing and commits or aborts, leaving the queue, the remaining transactions move forward in the queue, making C now eligible for execution as depicted in the third image of the queue. Notice that, in this case, it is guaranteed that at most 2 transactions can execute concurrently in the system, regardless of the number of replicas or of which replicas the transactions are local to. Also, the order in which transactions start to execute is not required to follow the

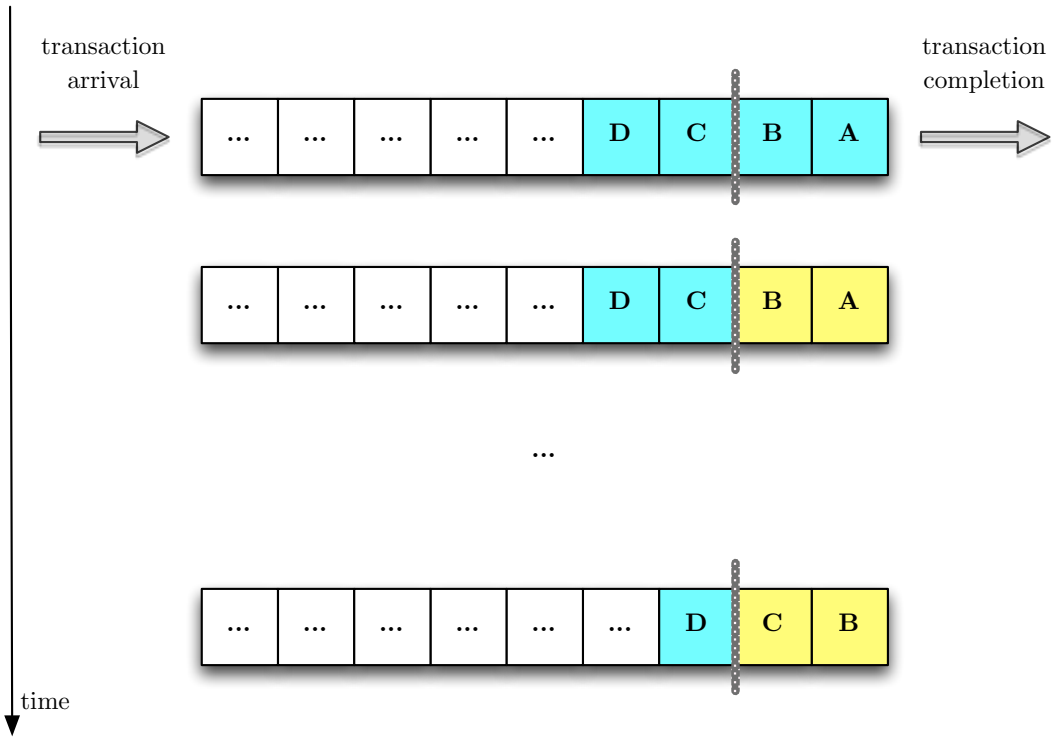


Figure 4.4: Threshold-based transaction eligibility for execution.

established commit order, allowing for asynchronous behaviour among replicas.

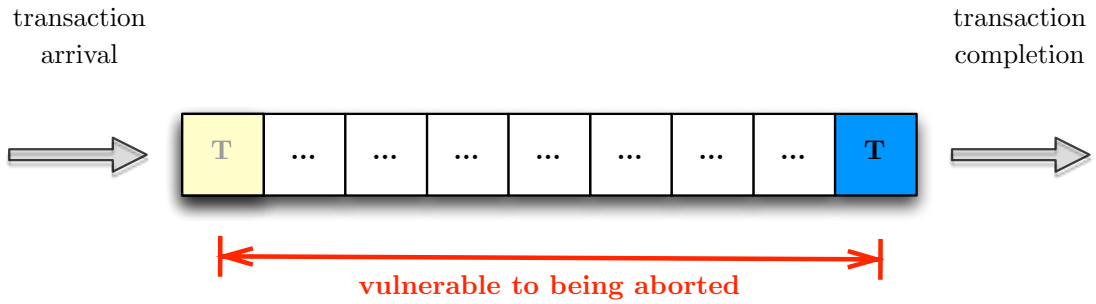
It is important to point out that introducing a finite delay before execution does not affect the correctness of the system: a similar schedule, with out-of-order execution, could occur simply as a consequence of the interleaving of threads in a database engine.

Under sufficient load, the threshold mechanism limits the number of transactions executing concurrently, thus decreasing the likelihood of conflicts occurring, and causes transaction execution to start at a later time than it originally would, thus decreasing the overall post-execution delay. This results in reducing the window during which each transaction is vulnerable to being aborted significantly. Figure 4.5 shows, for transaction T: its vulnerability window if executed immediately (Figure 4.5a; and if executed using the threshold mechanism (Figure 4.5b). The window shrinks significantly for the latter.

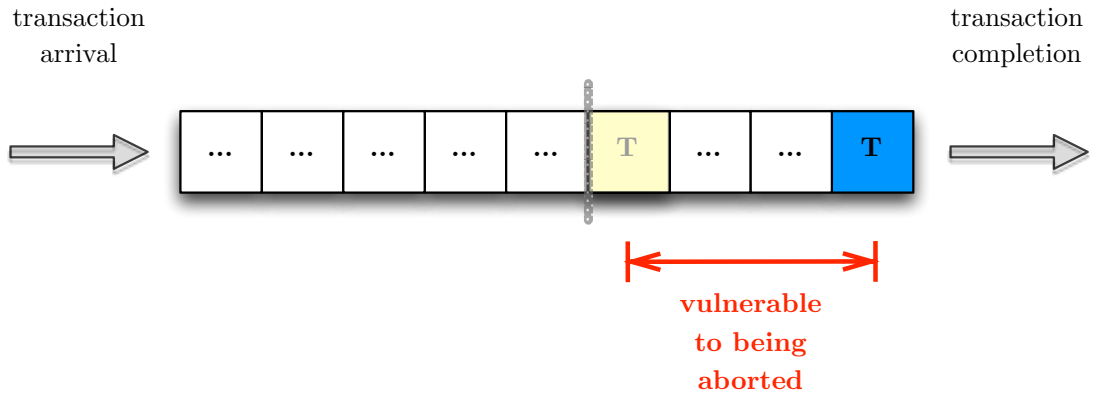
This fixed threshold mechanism simulates an admission control policy, in which the number of transactions allowed to execute concurrently is statically limited (Correia et al. 2008).

Algorithm 2: Fixed threshold throttling protocol algorithm.

```
1  t: transaction, Q: queue, not_executed, executing, executed, committing,
   committed, aborted : states ;
2  upon t is submitted
3  |  enqueue(Q,t);
4  |  t.state ← not_executed;
5  |  executeEligibleTransactions (Q);
6  |  return;
7  end
8  upon t is executed
9  |  t.state ← executed;
10 end
11 upon (t.state == executed ∧ head(t,Q))
12 |  if certified(t) then
13 |  |  t.state ← committing;
14 |  |  commit t;
15 |  else
16 |  |  abort t;
17 |  end
18 end
19 upon t is committed
20 |  t.state ← committed;
21 |  dequeue(Q,t);
22 |  executeEligibleTransactions (Q);
23 end
24 upon t is aborted
25 |  t.state ← aborted;
26 |  dequeue(Q,t);
27 |  executeEligibleTransactions (Q);
28 end
29 function executeEligibleTransactions (Q)
30 |  foreach t ∈ {t ∈ Q|t.state == not_executed ∧ t is beyond the
   |  threshold} do
31 |  |  execute t ;
32 |  end
33 |  return;
```



(a) Vulnerability window for some transaction for the baseline protocol.



(b) Expected vulnerability window for some transaction using the threshold throttling mechanism.

Figure 4.5: Effect of the threshold on transaction vulnerability.

4.2.1 Impact of Scheduling

Ideally, the threshold would be placed such that each transaction t completes execution just as it arrives at the head of the queue, minimizing the post-execution delay. Again, if the transaction reaches the head of the queue in either *not_executed* or *executing* states, it cannot be certified until it finishes. Because certification must occur in a conflict-equivalent order to the already established total order, transactions running late cannot be overtaken by others, thus impairing throughput.

A naïve approach would be to adjust the threshold simply by moving it one position back whenever a transaction reaches the head of the queue in the *not_executed* or *executing* states or one position forward whenever a transaction has to wait in the *executed* state or has been *aborted*. Such an adaptation mechanism, while simple, causes oscillation in the system, as the changes are too abrupt (Aström and Murray 2007).

A more evolved approach requires knowing (or estimating):

- how long it will take to execute the transaction (*i.e.* its execution latency) and
- how long it will take for the transaction to reach the head of the queue.

Transactions can have widely varying execution latency (*i.e.* duration), which should be considered when scheduling them: larger transactions should be executed earlier while smaller transactions should be executed later.

Let d_t be an estimate[1] of the duration of transaction t. Assume there is a constant factor relating it with the position of the threshold for that transaction (*input*).

Numbering the queue positions starting from the head of the queue with position 1, let

$$threshold_t = \lfloor input \cdot d_t \rfloor \tag{4.1}$$

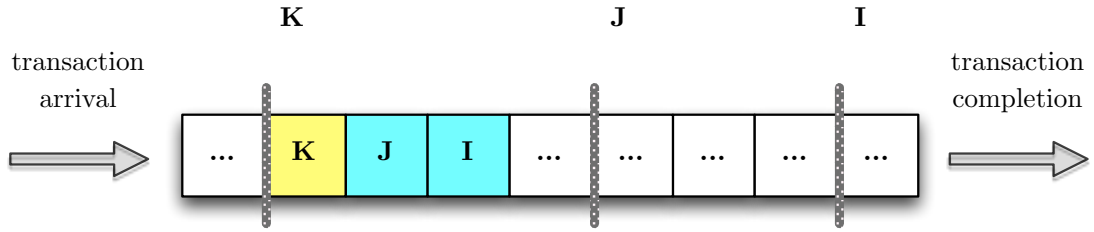
be the position (in the queue) after which transaction t will be executed.

Transaction t is scheduled to be executed when there are $threshold_t - 1$ or less transactions ahead of it in the queue, which is the same as placing a threshold for t in the queue at $threshold_t$ and executing t when it crosses it. For example,

[1] Details of how such an estimate can be obtained are discussed later on.

Transaction	Estimated Duration	Threshold Position
I	200 ms	4
J	50 ms	1
K	400 ms	8

(a) Assuming $input = 0.02$, threshold positions are calculated for each transaction.



(b) Execution thresholds for transactions with significantly different estimated durations.

Figure 4.6: Out-of-order execution with multiple thresholds.

assuming a static value of *input* (0.02), Figure 4.6 shows for transactions I, J and K, so ordered, where each threshold would be placed according to Equation 4.1. In the figure, transaction K has crossed the K-marked threshold, so its execution has begun, while J and I are still waiting in the *not_executed* state, despite preceding K in the commit order. Scheduling transactions based on their estimated duration enables out-of-order execution: a small transaction such as I will begin execution near the head of the queue, while a very large transaction as such K will be executed as soon (or almost as soon) as it is submitted.

Scheduling transaction execution in terms of the position in the queue allows the schedule to automatically adjust to changes in the overall system throughput: if it rises, a transaction t will progress faster in the queue and start executing sooner than with lower throughput. If this were not the case, system throughput would need to be explicitly considered when scheduling transactions. In particular, the schedule would need to be continuously updated as the estimate of how long it will take for transactions to reach the head of the queue changes with system throughput.

Applying Equation 4.1 to Algorithm 2 produces multiple lines: after being submitted, the transaction's execution is scheduled (Algorithm 3 line 5) using an

Algorithm 3: Threshold-per-transaction scheduling using a fixed *input* value, based on Equation 4.1.

```
1  t: transaction, Q: queue, not_executed, executing, executed, committing,
   committed, aborted : states, input : float ;
2  upon t is submitted
3  |  enqueue(Q,t);
4  |  t.state ← not_executed;
5  |  scheduleTransaction (t);
6  |  executeEligibleTransactions (Q);
7  |  return;
8  end
9  upon t is executed
10 |  t.state ← executed;
11 end
12 upon (t.state == executed ∧ head(t,Q))
13 |  if certified(t) then
14 |  |  t.state ← committing;
15 |  |  commit t;
16 |  else
17 |  |  abort t;
18 |  end
19 end
20 upon t is committed
21 |  t.state ← committed;
22 |  dequeue(Q,t);
23 |  executeEligibleTransactions (Q);
24 end
25 upon t is aborted
26 |  t.state ← aborted;
27 |  dequeue(Q,t);
28 |  executeEligibleTransactions (Q);
29 end
30 function executeEligibleTransactions (Q)
31 |  foreach
   |  t ∈ {t ∈ Q|t.state == not_executed ∧ t.position ≥ t.threshold} do
32 |  |  execute t ;
33 |  end
34 |  return;
35 function scheduleTransaction (t)
36 |  t.line = input ∗ t.estimatedDuration;
```

estimated transaction duration (Algorithm 3 lines 35, 36).

The *input* parameter provides a simple way to adjust how early transactions should be executed: for the same estimated duration, a higher value of *input* means that the transaction will be executed earlier than with a lower value.

4.2.2 Finding the Optimal *input*

While successive experiments with a given setting and a given workload can find an optimal throughput plateau matching a small range of *input* values, this is impractical for real systems.

Finding an appropriate *input* value without resorting to trial and error requires an adaptive mechanism that reacts to some measurement (or a set thereof) that reflects the relevant state of the system. Also, as system load changes, whether due to an increase in the number of clients (and consequently of the requests per second) or due an increase in overall transaction execution latency (*e.g.* as the database size grows) the optimal *input* value also changes.

The time a transaction spends queueing after being executed (*i.e.* its post-execution delay) depends on when it started to execute with respect to how long it takes for it to reach the head of the queue, so it is directly affected by the value of *input*.

For a given transaction, let q be the time it spent queueing (*i.e.* post-execution delay). Let $Qing$ be a weighted cumulative rolling average of q and $Qing_{opt}$ the optimal level of queueing for a system. An adaptive mechanism that reacts to the state of the queue can be defined using a proportional-integral-derivative (PID) controller (Aström and Murray 2007).

A PID controller algorithm features three terms: the proportional term, that depends on the magnitude of the current error value, the integral term that compensates for accumulated past error values and the derivative term that accounts for the rate of change of the error. Simply put, the error of the measured value (*sensor*) when compared to the desired value (*setpoint*) is used to update an input to the system (*input*) which will in turn impact the measured value, constituting the control feedback loop.

With $Qing$ as the sensor, $Qing_{opt}$ as the set point and *input* (from Equation 4.1) as the system input (Aström and Murray 2007) the feedback loop becomes:

$$
\begin{aligned}
error &= Qing_{opt} - Qing \\
P_{value} &= K_p \cdot error \\
I_{value} &= 0 \\
D_{value} &= 0 \\
input+ &= P_{value} + I_{value} + D_{value}
\end{aligned}
$$

K_p is referred to as *proportional gain*, a tuning parameter that adjusts the sensitivity of the controller, *i.e.*, the magnitude of the adaptation relatively to the magnitude of the error. Several methods exist for selecting an appropriate value for K_p, from manual tuning to methods based on heuristics (Aström and Hägglund 1984). In this particular instance, using just the proportional term for adaptation proved effective, as the system quickly converges to an appropriate stable state.

Intuitively, selecting the set point to target average duration would mean that there would always be a transaction ready to be certified and, consequently, that the rate at which transactions are certified is the same as the rate at which transactions arrive at the head of the queue. This scenario provides optimal throughput while minimizing the vulnerability window and, therefore, a minimal abort rate. Notice that depending on the how accurate the duration estimate is, selecting a set point of 0 might mean that several transactions would not finish its execution in time.

When scheduling transaction t using Equation 4.1, if t's actual duration is significantly larger that its estimate (d_t), then the measured queueing for transactions behind t in the queue will tend to increase as these will have to wait for t to finish its execution. Conversely, if t's actual duration is significantly smaller that d_t, t's measured queueing will be excessively large. In short, the error in estimating transaction duration can lead to increased queueing, causing the queue to grow as the increased queueing leads to decreasing *input* and increasing pre-execution delays in general.

If deemed necessary, instead of selecting the average duration as the set point, a higher percentile can be chosen from its cumulative distribution function: the higher the percentile of the chosen value, the higher the number of transactions

that will have completed execution as expected.

This adaptive mechanism not only allows the system to adapt to the current load by finding the appropriate *input* value, but it also enables it to adapt to changes in the workload, as long as there are sufficiently long periods of stability.

Algorithm 4 features the adaptive mechanism: whenever a transaction reaches the head of the queue and commits, its post-execution delay (*i.e.* queueing) is calculated and used to update the rolling average, $Qing$ (line 18); the difference between the updated value and the setpoint value ($Qing_{opt}$) is used as the measured error by the feedback adaptation mechanism to update the *input* value (lines 19, 34-37). The new *input* value is used to update the position of the threshold for each transaction, including those that have already been scheduled. Doing so ensures the effect of the adaptive mechanism is timely, allowing the system to swiflty adapt to significant changes in the state of the queue and consequently, the system. Consider the case of a decrease in system throughput because transactions are reaching the head of the queue before completing their execution. This means that the current value of input is too low, leading to an excessive pre-execution delay, causing the measured queuing to be approximately 0. If the thresholds for transactions already scheduled were not updated, this would only be corrected for new transactions being submitted, while the previous ones would needlessly have to wait before being executed, impairing throughput. Notice that the longer the queue (*i.e.* the more transactions there are in the system), the more this would adversely affect system performance. Also, the additional delay between action and effect introduced in the feedback loop, would result in accumulated errors that would require a more complex controller.

4.2.3 Estimating Transaction Execution Latency

Query plan optimization relies on estimating the cost of competing execution plans. This cost, however, is not a direct estimate of how long it will take a given plan to execute. Extrapolating a transaction's duration from the optimizer's output would require: tuning the planner to output cost in real time instead of a relative cost unit; and the whole transaction must be known to the optimizer before execution can start for it to output an estimate, thereby precluding interactive transactions.

Real-world OLTP applications usually run a set of transaction types, typically

Algorithm 4: Threshold-per-transaction scheduling with adaptation.

```
1  t: transaction, Q: queue, not_executed, executing, executed, committing,
   committed, aborted : states, Qing_opt : float ;
2  upon t is submitted
3  |  ...
4  end
5  upon t is executed
6  |  t.state ← executed;
7  end
8  upon (t.state == executed ∧ head(t,Q))
9  |  if certified(t) then
10 |  |  commit t;
11 |  else
12 |  |  abort t;
13 |  end
14 end
15 upon t is committed
16 |  t.state ← committed;
17 |  dequeue(Q,t);
18 |  update Qing;
19 |  ADAPT (Qing_opt,Qing);
   |  // Update thresholds.
20 |  foreach t ∈ {t ∈ Q|t.state == not_executed} do
21 |  |  SCHEDULETRANSACTION (t) ;
22 |  end
23 |  EXECUTEELIGIBLETRANSACTIONS (Q);
24 end
25 upon t is aborted
26 |  t.state ← aborted;
27 |  dequeue(Q,t);
28 |  EXECUTEELIGIBLETRANSACTIONS (Q);
29 end
30 function EXECUTEELIGIBLETRANSACTIONS (Q)
31 |  ...
32 function SCHEDULETRANSACTION (t)
33 |  t.threshold = input * t.estimatedDuration;
34 function ADAPT (setpoint, sensor)
35 |  error = setpoint − sensor;
36 |  P_value = K_p * error;
37 |  input+ = P_value;
```

encoded as stored procedures or in an application server. The position of the threshold for a transaction of type T, for example, can be calculated as

$$threshold_T = input \cdot d_T \tag{4.2}$$

where d_T is an estimate of the duration of transactions of type T, calculated online using a cumulative rolling average. Furthermore, the technique for conflict class extraction presented in Section 3.2.1 can be used to define transaction types according to the conflict classes accessed by transactions.

Still, AJITTS can be implemented without this simplification, computing a threshold for each individual transaction as long as an individual estimate can be provided.

Algorithm 3 can be implemented using Equation 4.2 becoming Algorithm 5: $TTypes$ is introduced as the set of possible transaction types and the thresholds are calculated using the fixed *input* value and the current estimate of average duration for that type (lines 37-41). Thresholds are updated whenever the estimates for average duration per type are updated (lines 10, 11). Instead of being scheduled individually, transactions that are found to be beyond the respective threshold whether because the threshold moves (line 12) or as transactions advance in the queue (lines 5, 25, 30) are executed.

As discussed, the set point should be chosen taking into consideration the distribution of the duration of all write transactions. A simple way to estimate the distribution is to sample transaction duration from the running system during a training period, as long as it can be assumed that average transaction duration is characteristic of the workload. If this assumption cannot be made because the workload changes, the initial estimate can be further improved by online sampling.

In Algorithm 6, D is introduced as the value corresponding to chosen percentile of transaction duration to be used as the set point. A threshold is calculated for each type of transaction in $TTypes$ based on the current estimate of average duration for that type (lines 33 to 37) and updated whenever a transaction of that type finishes execution (line 8) or whenever *input* changes (line 20). Eligible transactions are executed when the thresholds move (lines 9, 21) or as transactions advance in the queue (lines 9, 21, 26) . The estimate D is updated before re-calculating *input* (lines 18, 19).

Algorithm 5: Threshold-per-type scheduling with a fixed *input* value.

```
1  t: transaction, Q: queue, not_executed, executing, executed, committing,
   committed, aborted : states, TTypes: set, input: float;
2  upon t is submitted
3  |  enqueue(Q,t);
4  |  t.state ← not_executed;
5  |  executeEligibleTransactions (Q);
6  |  return;
7  end
8  upon t is executed
9  |  t.state ← executed;
10 |  update t.type.estimatedDuration;
11 |  updateThresholds ();
12 |  executeEligibleTransactions (Q);
13 end
14 upon (t.state == executed ∧ head(t,Q))
15 |  if certified(t) then
16 |  |  t.state ← committing;
17 |  |  commit t;
18 |  else
19 |  |  abort t;
20 |  end
21 end
22 upon t is committed
23 |  t.state ← committed;
24 |  dequeue(Q,t);
25 |  executeEligibleTransactions (Q);
26 end
27 upon t is aborted
28 |  t.state ← aborted;
29 |  dequeue(Q,t);
30 |  executeEligibleTransactions (Q);
31 end
32 function executeEligibleTransactions (Q)
33 |  foreach t ∈ {t ∈ Q|t.state == not_executed ∧ t.position ≥ t.type.line}
   |  do
34 |  |  execute t ;
35 |  end
36 |  return;
37 function updateThresholds ()
38 |  foreach type ∈ TTypes do
39 |  |  type.threshold = input ∗ type.estimatedDuration;
40 |  end
41 |  return;
```

Algorithm 6: Threshold-per-type scheduling with adaptation.

```
1  t: transaction, Q: queue, not_executed, executing, executed, committing,
   committed, aborted : states, D : float, TTypes: set;
2  upon t is submitted
3  |  ...
4  end
5  upon t is executed
6  |  t.state ← executed;
7  |  update t.type.estimatedDuration;
8  |  updateThresholds ();
9  |  executeEligibleTransactions (Q);
10 end
11 upon (t.state == executed ∧ head(t,Q))
12 |  ...
13 end
14 upon t is committed
15 |  t.state ← committed;
16 |  dequeue(Q,t);
17 |  update Qing;
18 |  update D;
19 |  adapt (D,Qing);
20 |  updateThresholds ();
21 |  executeEligibleTransactions (Q);
22 end
23 upon t is aborted
24 |  t.state ← aborted;
25 |  dequeue(Q,t);
26 |  executeEligibleTransactions (Q);
27 end
28 function executeEligibleTransactions (Q)
29 |  foreach
   |  t ∈ {t ∈ Q|t.state == not_executed ∧ t.position ≥ t.type.threshold} do
30 |  |  execute t ;
31 |  end
32 |  return;
33 function updateThresholds ()
34 |  foreach type ∈ TTypes do
35 |  |  type.threshold = input ∗ type.estimatedDuration;
36 |  end
37 |  return;
38 function adapt (setpoint, sensor)
39 |  error = setpoint − sensor;
40 |  P_value = K_p ∗ error;
41 |  input+ = P_value;
```

4.3 Summary

Although increasingly popular and often used, optimistic concurrency control may lead, with more demanding workloads, to a large number of conflicts and aborted transactions. This endangers fairness and reduces usable throughput. Previous attempts at tackling this problem required workload-specific configuration and would still impact peak throughput.

With AJITTS, the adaptive just-in-time transaction scheduler, we provide a solution that does not require workload specific configuration and adapts in runtime to current workload and resource availability conditions. This is achieved by delaying transaction execution, for each transaction individually, based on the estimated time to complete and current queueing within the system.

Chapter 5

Evaluation

The approach proposed in Section 4.2 is evaluated using a simple event-driven simulator that enables a profound analysis of each aspect of scheduling and concurrency control of replication protocols. This chapter describes how the simulation model implements the system model defined in Section 4.1, the workload that is used to run the simulation, the impact scheduling parameters have on behaviour and an evaluation of AJITTS performance when compared to the baseline protocol.

5.1 Simulation Model

The simulation model implements the system model presented in Section 4.1. Transaction lifecycle events are implemented as events in simulated time and inserted in the event list, ordered by timestamp. Simulated time progresses discretely through event timestamps. Event types are: *START* for transaction submission; *EXEC_START* when the transaction enters the *executing* state; *EXEC_END* when the execution is finished and the transaction progresses to the *executed* state; *CERT* equivalent to entering *committing*; *COMMIT* and *ABORT* when the transaction is committed or aborted, respectively. In the beginning of the simulation, the event list consists of all transactions' *START* events. Simulated time starts at the earliest event timestamp, *i.e.*, the first event to be consumed.

Implicit certification is simulated by detecting conflicts between the transaction being committed and transactions in either *executing* or *executed* states, as

transaction write sets are known (Algorithm 7, lines 14, 21 to 27).

Algorithm 7: Simulated implicit certification with commit-time conflict detection.

```
1  t: transaction, Q: queue, not_executed, executing, executed, committing,
   committed, aborted : states ;
2  upon t is submitted
3  |  ...
4  end
5  upon t is executed
6  |  ...
7  end
8  upon (t.state == executed ∧ head(t,Q))
9  |  ...
10 end
11 upon t is committed
12 |  t.state ← committed;
13 |  DETECTCONFLICTS (Q,t);
14 |  dequeue(Q,t);
15 |  ...
16 end
17 upon t is aborted
18 |  ...
19 end
20 function DETECTCONFLICTS (Q,t)
21 |  foreach
   |  {t' ∈ Q|t' ≠ t ∧ (t'.state == executing ∨ t'.state == executed)} do
22 |  |  if t'.ws ∩ t.ws ≠ ∅ then
23 |  |  |  abort t';
24 |  |  end
25 |  end
26 |  return;
27 ...
```

As discussed in Section 4.2, the set point should be chosen taking into consideration the distribution of transaction duration. In the results presented here, the set point used in AJITTS is the value of the average global transaction duration of the given workload.

5.2 Workload

In order to obtain realistic write sets for certification, the simulation workload is based on TPC-E. The simulator consumes execution traces obtained by running a TPC-E like benchmark on a centralized MySQL[1] database and then parsing the resulting binlog to generate the workload. The simulator uses the following information from the binlog: the timestamps at which each transaction started, how long it took to execute each transaction and each transaction's write sets.

The load generated by serial runs of TPC-E over the same database can be parallelized by creating unique identifiers for each transaction and by manipulating timestamps making these relative to a reference instant. As a result, the load applied to the protocol under test can be easily scaled. Also, the applied load is not limited by resource constraints on the original MySQL database: there is no limit on the number of load units that can be applied in parallel.

The transaction duration values extracted from the binlog reflect the penalty introduced by synchronization and locking in the MySQL engine when the original benchmark is executed. A correction factor (β) can be calibrated by running the traces through the simulator with optimistic scheduling, without admission restrictions and without re-execution, *i.e.*, executing transactions immediately upon submission, chosen such that the abort rate is close to 1%. The reason for this is that the sequence of transactions in the binlog is implicitly proved to be conflict-free with the original values for transaction duration.

Let dur'_t be the duration extracted from the binlog for transaction t. The respective value to be used in the simulation is

$$dur_t = \beta * dur'_t \tag{5.1}$$

The value of the correction factor depends on the benchmark load induced on MySQL. Therefore, the β used in the simulation is independent of the number of parallel traces used to fuel the simulator, as long as the load induced on MySQL by each benchmark run was about the same. If using another set of traces, the correction factor must be recalculated. β was found to be 0.2 for the traces used for evaluation.

Because of the way the load is scaled, the dilation effect of transaction duration

[1] `http://www.mysql.com`

as load increases is not considered. In any case, with a system functioning at or below nominal capacity, this effect should be negligible. Notice that this is not a limitation of AJITTS.

5.3 Impact of Scheduling Parameters

Using the event-driven simulator with Algorithm 5 (and simulated implicit certification as presented in Algorithm 7) Figure 5.1 shows the latency breakdown for a particular workload (400 clients) for different fixed values of the *input* parameter, *i.e.*, without adaptation. On the right hand side, transactions are scheduled early, thus decreasing the amount of time spent in the *not executed* state, shown in blue. In fact, an extreme setting of the parameter causes transactions to be scheduled for immediate execution, equivalent to the baseline protocol of Algorithm 1(again, with simulated implicit certification as presented in Algorithm 1): the pre-execution delay is negligible, while spending a sizeable amount of time in the *executed* state. On the left, transactions are scheduled later, thus waiting an increased amount of time before execution, but waiting very little as *executed* (in orange). In particular, the average vulnerability window per transaction decreases from $568ms$ in the far right to $179ms$ in the far left, considering that the *input* parameter does not have an impact in the execution latency (in yellow). As expected, overly delaying transaction execution has an impact on total latency.

Figure 5.2 shows a complete set of statistics for a broader range of input values for three workloads that differ only in the number of concurrent clients submitting transactions. Besides impacting end-to-end latency, the *input* parameter has an effect on throughput and the ratio of aborted transactions, leading to the following trade off:

- On the left, with a larger pre-execution delay, transactions arrive at the head of the queue but are not yet fully executed, thus stalling the queue and leading to reduced throughput. However the small post-execution delay leads to a reduced number of concurrency-related aborts.

- On the right, transactions are executed fairly ahead of time, thus avoiding stalling the queue. However, by having been started early they become concurrent with a larger number of transactions which leads to an increased

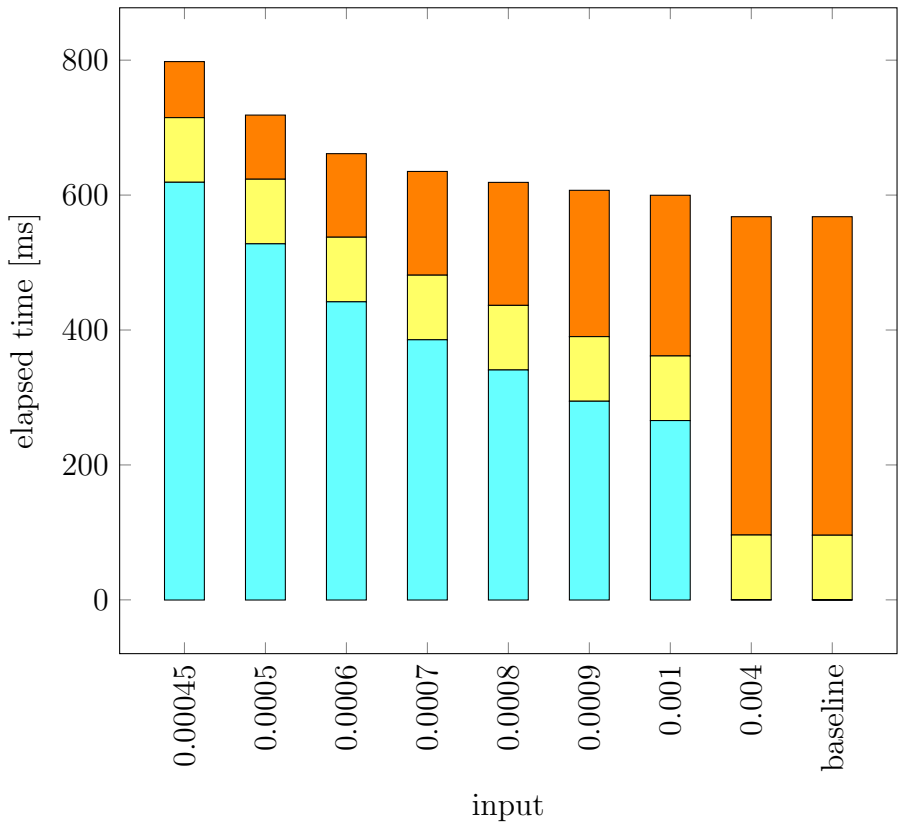


Figure 5.1: Latency breakdown for different fixed values of the scheduler parameter: pre-execution delay (blue), execution latency (yellow), and post-execution delay (orange), *i.e.*, time spent in the *not executed*, *executing* and *executed* states respectively.

number of concurrency-related aborts.

Notice that, for example, if *input* is between $0.4 \cdot 10^{-3}$ and $0.9 \cdot 10^{-3}$ for 800 clients, throughput is sub-optimal because transactions are being executed too late (Figure 5.2). Also, for the same workload, the abort ratio steadily rises as *input* increases, until for a large enough value of *input*, it stabilizes. For example, with 200 clients, the abort ratio stabilizes at 5%, for *input* larger than $1 \cdot 10^{-3}$. This happens because after this point almost all transactions are executed as soon as they are submitted, reducing to the baseline protocol.

The bottom-right chart of Figure 5.2 shows the ratio between the average queueing and the average duration of all transactions. By comparison with the top-left chart, showing throughput, the input values that achieve optimal throughput in the top-left chart match those for which the ratio in the bottom-left chart is approximately 1. This confirms the intuition presented in Subsection 4.2.2 that system behaviour is optimal when the average queueing is similar to the average duration and that the latter should be used as the set point for adaptation.

Figure 5.3 shows how the system behaves for the same number of clients but with different resource availability. Lower resource availability is simulated by increasing transaction execution latency. Notice that the likelihood of conflict rises with the increase in transaction execution latency. Still, the same trade-off holds.

In Figure 5.2, the optimal *input* value for 200, 400 or 800 clients is respectively $0.19 \cdot 10^{-3}$, $0.43 \cdot 10^{-3}$ and $1 \cdot 10^{-3}$; in Figure 5.3, the optimal *input* value for duration D, D*2.5 and D*4 is respectively $0.43 \cdot 10^{-3}$, $0.5 \cdot 10^{-3}$ and $0.6 \cdot 10^{-3}$.

The plateaus are wide: using an *input* value of $1 \cdot 10^{-3}$ with 200 clients, *e.g.*, instead of the optimal value, yields a negligible impact on throughput. However, as the load increases either by serving more clients or by processing larger transactions, the right-hand slope of the throughput curve becomes more and more accentuated, increasing the toll on throughput if the *input* value used in not optimal. This behaviour is to be expected as excessive load leads to an increase in the abort ratio due to higher concurrency (Figures 5.2 and 5.3).

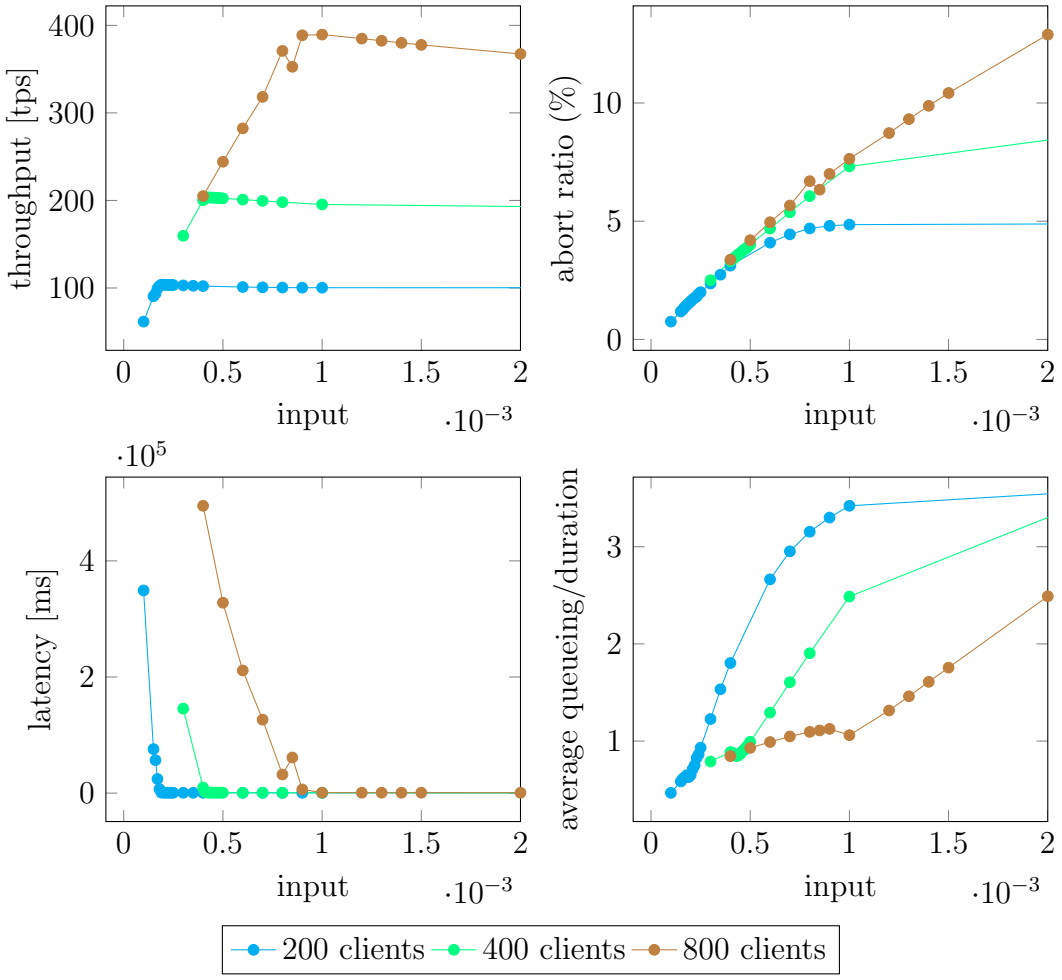


Figure 5.2: Effect of the *input* value on throughput, the abort ratio, transaction latency and on the ratio between average transaction queueing and average duration for different numbers of clients.

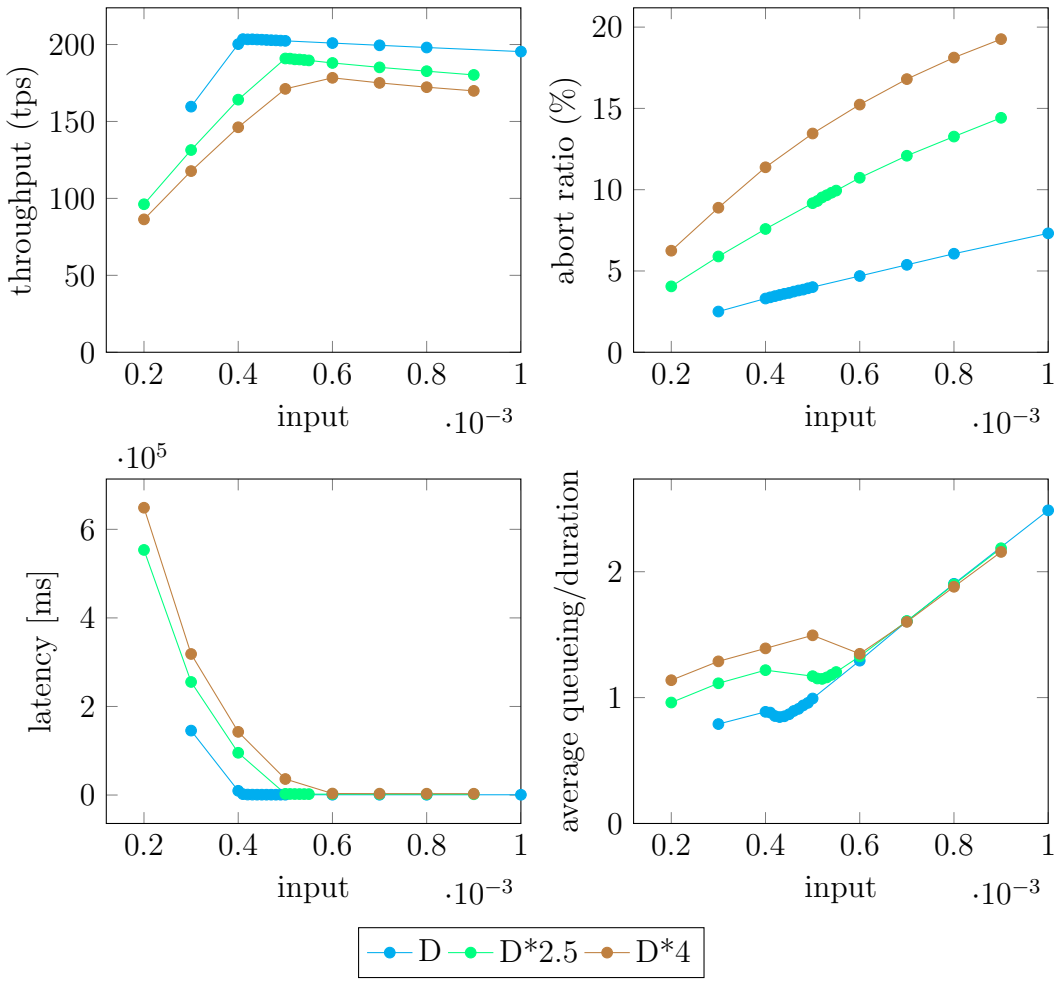


Figure 5.3: Effect of the *input* value on throughput, the abort ratio, transaction latency and on the ratio between average transaction queueing and average duration for different distributions of transaction duration (*i.e.* transaction execution latency).

5.4 Performance

We compare AJITTS with the baseline protocol described in Algorithm 1 that executes each transaction as soon as it is submitted. Figure 5.4 compares the baseline protocol and AJITTS in terms of throughput and aborts for three workloads that differ only on the number of concurrent clients submitting transactions. Notice that even though AJITTS introduces delays on transaction executions, throughput is not only not adversely affected, but actually improved. Also, AJITTS clearly succeeds in significantly reducing the abort rate. In fact, a clear trend of further improvement can be observed in both charts as the load increases.

Figure 5.5 shows how the threshold positions per transaction type evolve during a run with a particular workload. Threshold positions are updated whenever the estimates for execution duration change or whenever the adaptation input parameter changes. The position of the threshold for each transaction type converges quickly: the amplitude of the variation stabilizes after considerably few updates. In particular, TU transactions actually consist of three different types of sub-transactions as described in Section 3.1: the variability of the duration of trade update transactions is mirrored in the variation of the position of the threshold for this type of transaction. Notice that TU transactions, significantly larger compared to other transactions, are scheduled much earlier than the other types of transactions. Figure 5.5 also shows the cumulative distribution function of the measured queueing (q) aggregated by transaction type, which is a result of the position of the thresholds.

Figure 5.6 shows how the different average durations (in yellow) influence the pre-execution delay (in blue) when using AJITTS: again, TU transactions (TU-AJITTS) are scheduled much earlier than others, while MF transactions (MF-AJITTS), for instance, are only executed nearer the head of the queue. When comparing the results regarding, for example, MF transactions, the average time during which these are vulnerable to being aborted much smaller using AJITTS ($110ms$) than using the baseline protocol ($562ms$). This is also the case for TR and TO transactions.

Notice that the average queueing of TU transactions actually increases using AJITTS. This is a consequence of choosing average duration as the set point for adaptation. Equation 4.2 causes the average post-execution delay for each

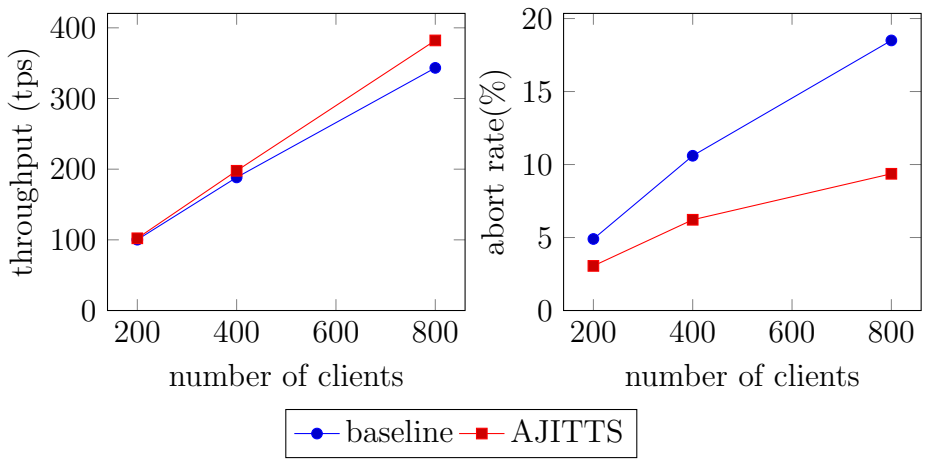


Figure 5.4: Throughput and abort rate using AJITTS instead of the baseline protocol in scenarios with different numbers of clients.

transaction type to approximate each type's transaction duration, as evident in Figure 5.6. Because TU transactions are, on average, much larger than others, this results in an increase of the average queueing for TU transactions. In short, this is the cost of the effort of ensuring there is always some transaction ready to be committed. Also, while, due to its relative size, 84% of TU transactions are executed immediately upon submission, it is still possible for the queue to grow so that some TU transactions, when submitted are still below the threshold, resulting in the measured average pre-execution delay. This is evidence that AJITTS handles peaks of increased load correctly. As expected, the net effect is still a reduced abort rate.

Considering different duration distributions shapes the workload: higher durations simulate less available resources and vice-versa. Figure 5.7 shows how AJITTS leverages available resources significantly better than the baseline protocol. In particular, the less available resources, the more the baseline protocol's throughput decreases relatively to AJITTS.

5.5 Summary

Chapter 4 presented an adaptive transaction scheduler that leverages transaction execution estimates and the measured level of queueing in the system to min-

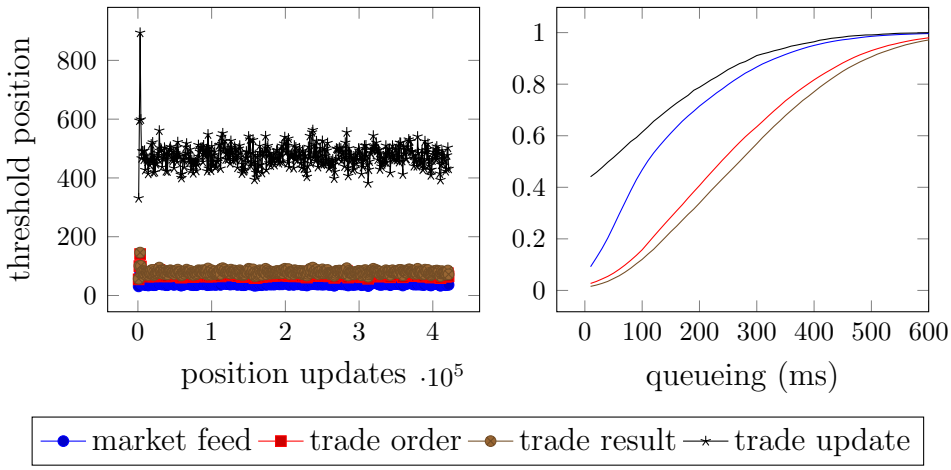


Figure 5.5: Evolution of the position of the threshold during a particular run.

imize the number of aborted transactions. Evaluating AJITTS in a simulated environment provided insight into how each decision in the design of the adaptive mechanism affected the behaviour of the protocol. The ability to manipulate the load made it possible to test the algorithm under very high load and with varying probabilities of conflict.

AJITTS was evaluated using a simulation model driven by traces from TPC-E running on MySQL, demonstrating that it clearly outperforms the baseline protocol. In fact, in addition to reduced aborts, it actually improves peak throughput even if it throttles transaction execution. This is a consequence of using available resources better.

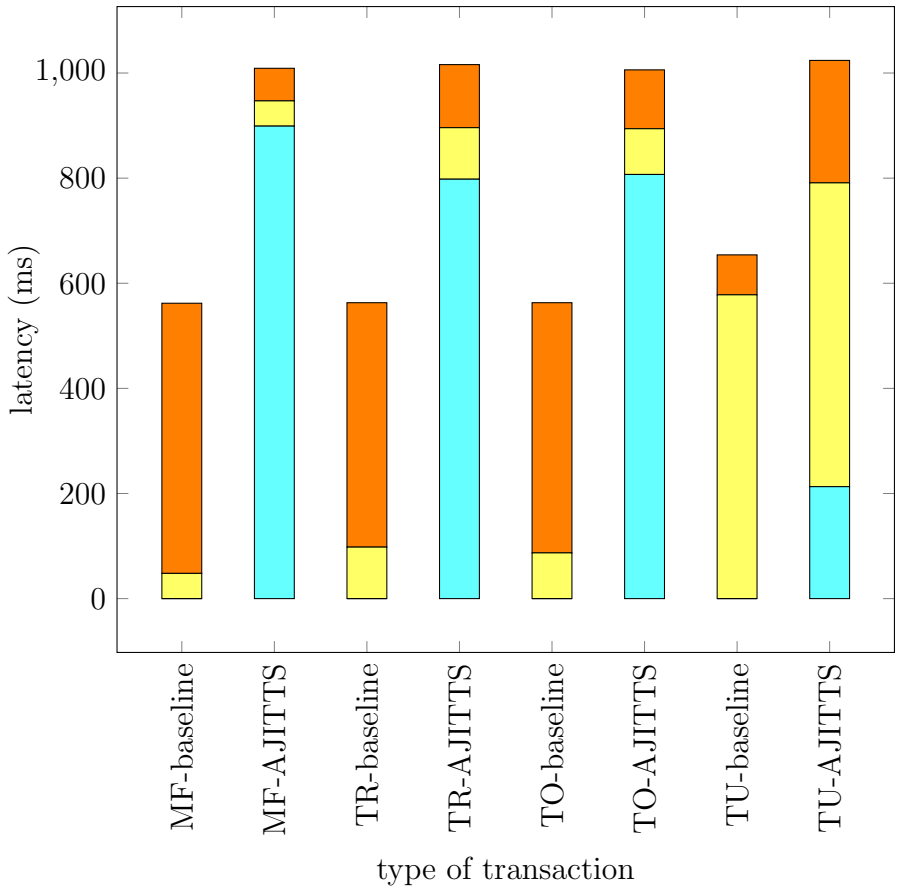


Figure 5.6: Latency breakdown using AJITTS and the baseline protocol: Pre-execution delay (blue), execution latency (yellow), and queueing for certification (orange). Columns MF-AJITTS, TR-AJITTS, TO-AJITTS and TU-AJITTS refer to an execution of the AJITTS protocol, while the others refer to an execution of the baseline protocol.

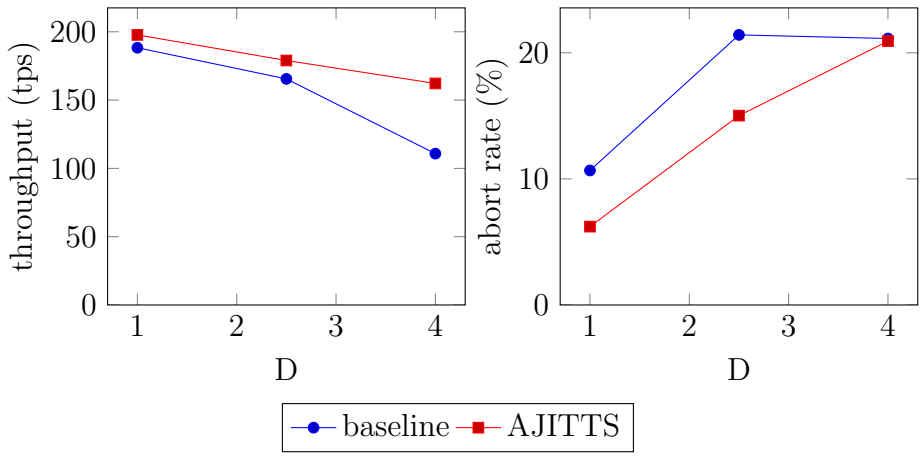


Figure 5.7: Throughput and abort rates for the baseline protocol and AJITTS for different duration distributions.

Chapter 6

Implementation

A full-featured implementation of AJITTS interacting with an actual RDBMS, capable of handling multiple replicas and running a TPC-E like benchmark is presented in this chapter. The goal of this effort is to show that: (1) AJIITS can be implemented in a real system; and (2) that simply scheduling transactions to execute sooner or later has the desired effect on the global queueing average. First, the components of the environment on which AJITTS is implemented, are described, followed by how the system model can be instantiated in this environment, focusing on novel features of the certification mechanism. Then, the workload is described, detailing the particularities of running TPC-E in the system. Finally, results are presented.

6.1 Model

This implementation of AJITTS is based on the replicated queue model built using a group communication protocol. Each replica has a copy of the entire database. Transactions can be submitted to any replica and are executed at the replica to which they were submitted. Each replica independently schedules its transactions and can independently certify all transactions. Aborted transactions are not automatically re-executed and the decision to resubmit them is left to the client. While in the simulation model transaction duration is considered to be independent of system load, that is not the case in this setting. The higher the load (*i.e.* more transactions executing concurrently or larger transactions), the longer it should take a given transaction to execute.

For the sake of simplicity, each transaction is considered to be implemented as a set of stored procedures, where the name of the first stored procedure to be called is sufficient to classify the transaction according to its type. However, the only requirement is that the first statement of the transaction enables it to be classified. For example, a dummy statement, which can even be read-only, can be used to hint at the type of the transaction.

Two kinds of interactive transactions are considered: those in which the time spent waiting for the user exhibits low variance, in which case a useful transaction duration estimate can be calculated and used by the adaptive mechanism; and those for which time spent waiting for the user varies significantly, where a sufficiently accurate estimate for transaction duration cannot be calculated. While the former are fully supported, supporting the latter might destabilize the adaptive mechanism.

It is assumed that the possible types of transaction that execute in a given system are known, either determined directly or using the method presented in Subsection 3.2.1.

Transaction scheduling is based on a line per type of transaction as in Equation 4.2. Transaction duration is estimated per type, using an online cumulative rolling average as described on Subsection 4.2.3.

While 1-copy snapshot isolation is assumed, serializability is also supported simply by changing the criteria for conflict detection to also consider transaction read sets and by sending both read and write sets to other replicas (as in the DBSM protocol (Pedone et al. 2003)). Some protocols offer the option of actively executing selected transactions when the read and/or write sets are expected to be significantly large, in an effort to reduce bandwidth consumption (Correia Jr et al. 2007). In others, read sets are not disseminated at the cost of allowing only the replica that locally executes the transaction to certify it (Kemme and Alonso 2000),

Figure 6.1 shows transaction states and possible transitions for local transactions. Additional states, when compared to Figure 4.1, are related to tasks that were immediate in the simulation but cannot be considered as such in an actual implementation. For example, while transaction order is determined *a priori* in the simulated environment, in this environment a transaction can only start *executing* after the its place in the order has been established, either by the group

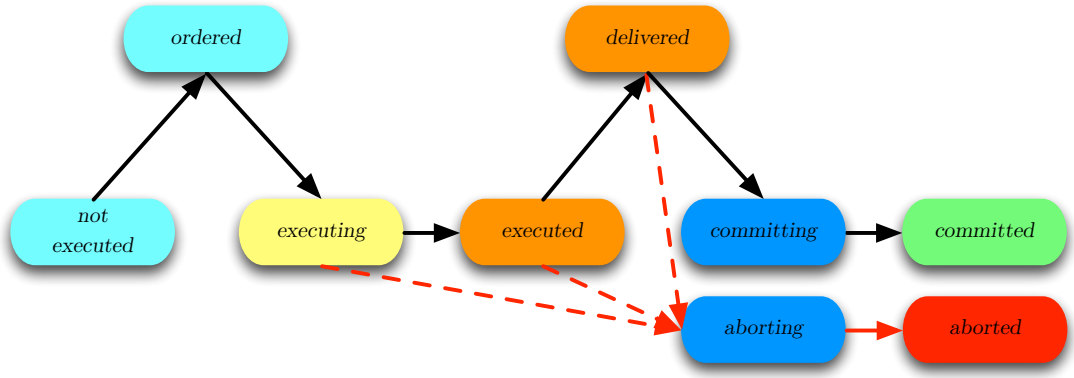


Figure 6.1: Transaction states and allowed transitions for local transactions.

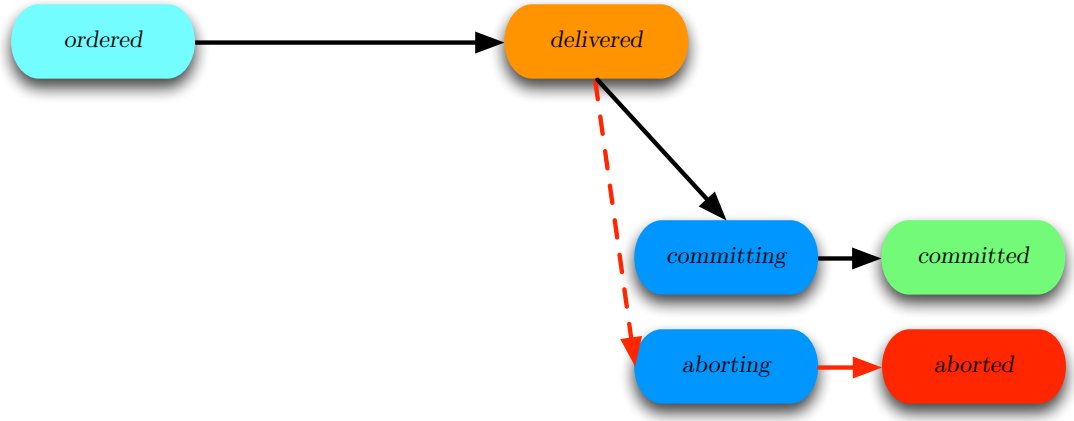


Figure 6.2: Transaction states and allowed transitions for remote transactions.

or some sequencer. Still, notice that by collapsing the states that are coloured the same, the graph is the same as that in Figure 4.1.

Starting in the *not executed* state, transactions transition to the *ordered* state once their place in the global total order has been established. When the execution starts, transactions enter the *executing* state, progressing to the *executed* state when finished; once it can be guaranteed that the necessary information about the transaction (read and/or write sets) will be received by other replicas, the transaction enters the *delivered* state. Once there is no preceding *delivered* transaction that can potentially conflict with it, if the transaction can be serialized with previously committed concurrent transactions it progresses to the *committing* state and once the commit is complete to *committed*; if not, it must be aborted, entering the *aborting* state and once complete, *aborted*. While in

executing, *executed* and *delivered* states a transaction can be aborted by other transactions, when a transaction is being certified, it can only be aborted by the certification mechanism.

Figure 6.2 shows transaction states and allowed transitions for remote transactions. A replica only becomes aware of a remote transaction when it is being ordered, which is why *not executed* is not shown. Similarly, the *executing* and *executed* states are omitted.

Starting in the *ordered* state, the transaction progresses to the delivered state when its information (again, read and/or write sets) is received. Once the transaction can be certified it progresses to *committing* and later *committed* if it can be committed, otherwise entering *aborting* and later progressing to the *aborted* state. As local transactions, remote transactions can be aborted in the *delivered* by other transactions but, during certification, only by the certification mechanism.

6.2 Details

The ESCADA replication server, developed in the context of the GORDA project (Correia Jr et al. 2007; Carvalho et al. 2007), provides a pluggable replication framework as shown on Figure 6.3. The goal of implementing AJITTS using ESCADA is that the latter handles interfacing with database engines and with group communication protocols abstracting most details of the particular implementations chosen. In short, ESCADA provides the following database capabilities through handlers, defined as a part of the replication protocol, to be called at specific stages of transaction processing (*e.g.* when a transaction has been submitted but before its execution starts, or when it has finished its execution and it is ready to commit, but before it does):

- to pause or continue transaction execution according to the design of the replication protocol;
- to access transaction specific information (*e.g* time stamps, read and write sets);
- to inspect, modify or inject SQL statements;

- to allow the replication protocol to decide whether a given transaction should commit or abort;

ESCADA interacts with the RDBMS through a generic API, the GAPI, that provides the capabilities itemized above as well as the ability to inject updates originating from remote transactions. By design, the capabilities provided by the GAPI are an abstraction of what is common in most database engines. Because AJITTS can be implemented over this abstraction, it can also be implemented in most relational database engines.

A simple interface to group communication is also provided by ESCADA to replication protocol implementations, reflecting different delivery guarantees offered by the chosen group communication protocol. For example, the standard group communication framework used in the GORDA project, APPIA (Miranda et al. 2001) offers a single *send* primitive that naturally sends the specified message to the group and three message delivery primitives:

optimistic guaranteed to be delivered by all correct processes but not necessarily in this position in the total order (used by protocols that exploit early optimistic delivery);

regular guaranteed to be delivered in this position in the total order by all correct processes and

uniform guaranteed to be delivered in this position in the total order even by faulty processes.

The framework provides the building blocks for implementing several replication strategies: primary-backup, state-machine (*i.e.* active replication) and certification-based replication (Correia Jr et al. 2007).

The sequence diagram in Figure 6.4 shows how a transaction is processed in the replicated system:

Step 1: The client connects to the database and starts a transaction, which triggers a handler invocation on the coordination kernel.

Step 2: The client submits a statement to the database, which before being executed triggers another handler invocation in the coordination kernel. By parsing the statement's SQL code, the coordination kernel is able

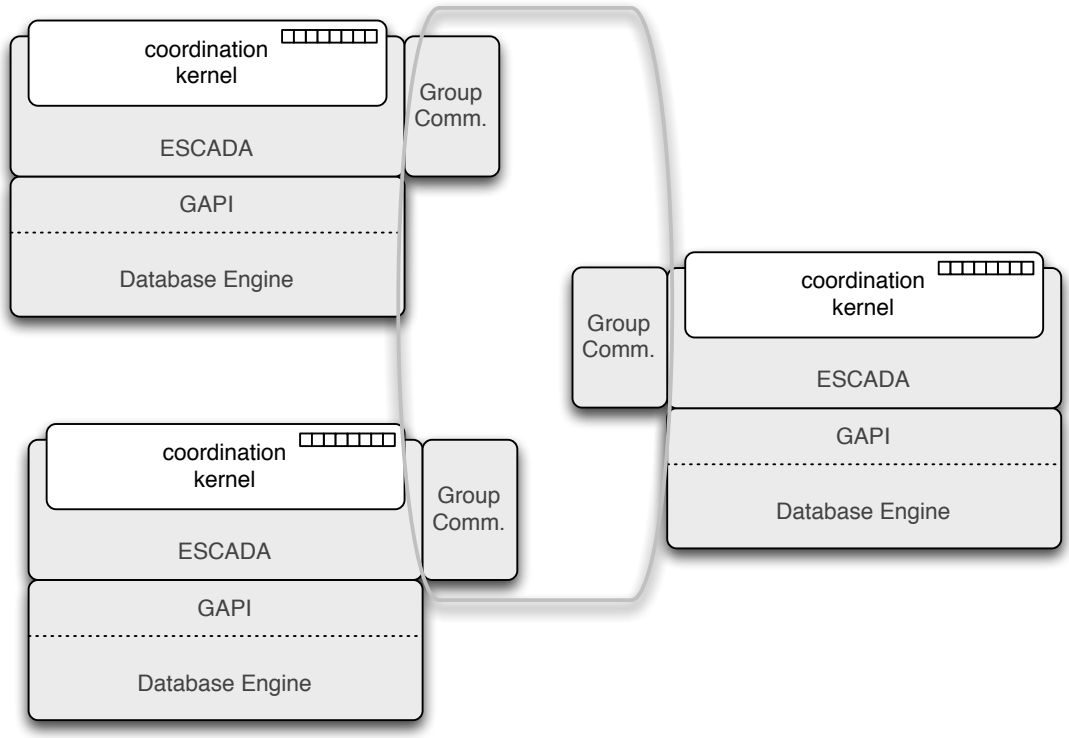


Figure 6.3: ESCADA stack.

to determine the transaction's type (*i.e.* to classify it). At this point the transaction enters the *not executed* state. The object that reflects the transaction is then sent to the group using a total order broadcast primitive.

Step 3: Upon uniform delivery, each replica's coordination kernel enqueues the transaction in its instance of the replicated queue: the transaction is local at the originating replica and remote at all others. The transaction enters the *ordered* state.

Step 4: When the coordination kernel at the originating replica finds the transaction is eligible for execution, it notifies the local DBMS to continue, executing the transaction, which enters the *executing* state.

Step 5: When the transaction sucessfully finishes executing, but before it is allowed to commit, the corresponding coordination kernel handler is invoked. At this point the transaction enters the *executed* state and the transaction's write set is sent to the group.

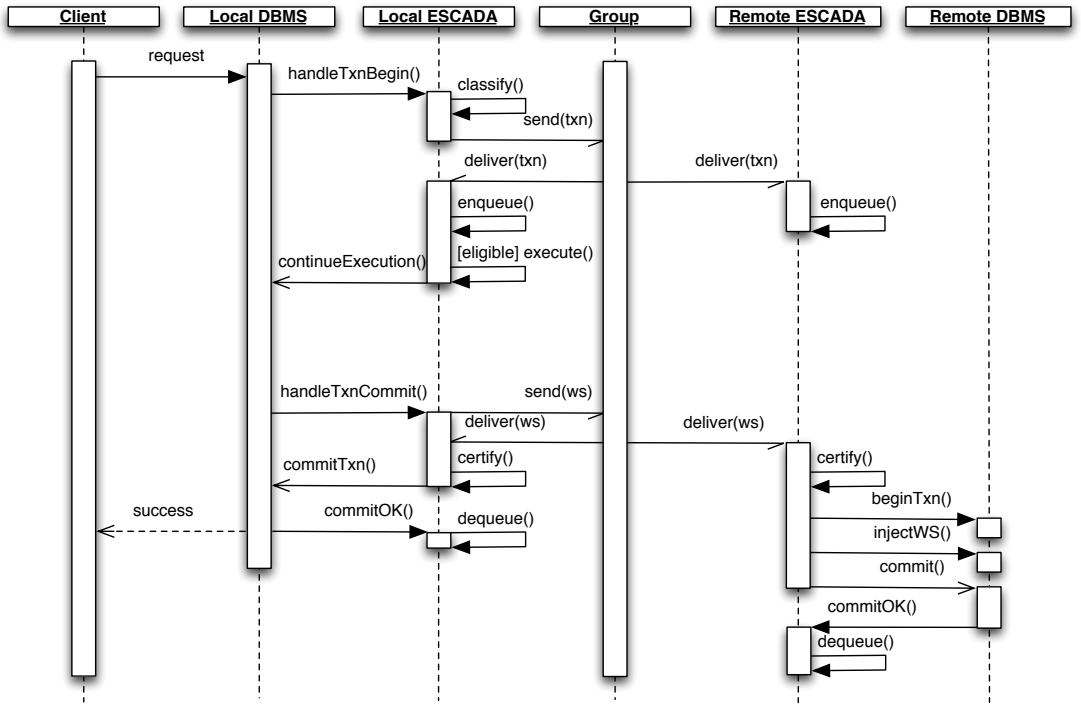


Figure 6.4: Sequence diagram for the ESCADA implementation of AJITTS.

Step 6: When the write set is delivered, the transaction enters the *delivered* state. Each replica can certify it independently and the decision is guaranteed to be the same at each replica because every replica knows every transaction's write set and does so in the established order.

Step 7: If the decision is to commit the transaction, the database enters the *committing* state and the local database is notified by the coordination kernel to continue the transaction's processing, committing it. Each remote replica's coordination kernel starts a transaction at its database, injects and applies the transaction's write set and commits it.

Step 8: Each database notifies the corresponding coordination kernel that the transaction has been committted sucessfully, entering the *committed* state, upon which the transaction is removed from the replicated queue.

However, transactions can be aborted: either because the database's local concurrency control mechanism determined a transaction cannot commit (Figure 6.5) or because it could not be certified (Figure 6.6).

In the case of the former:

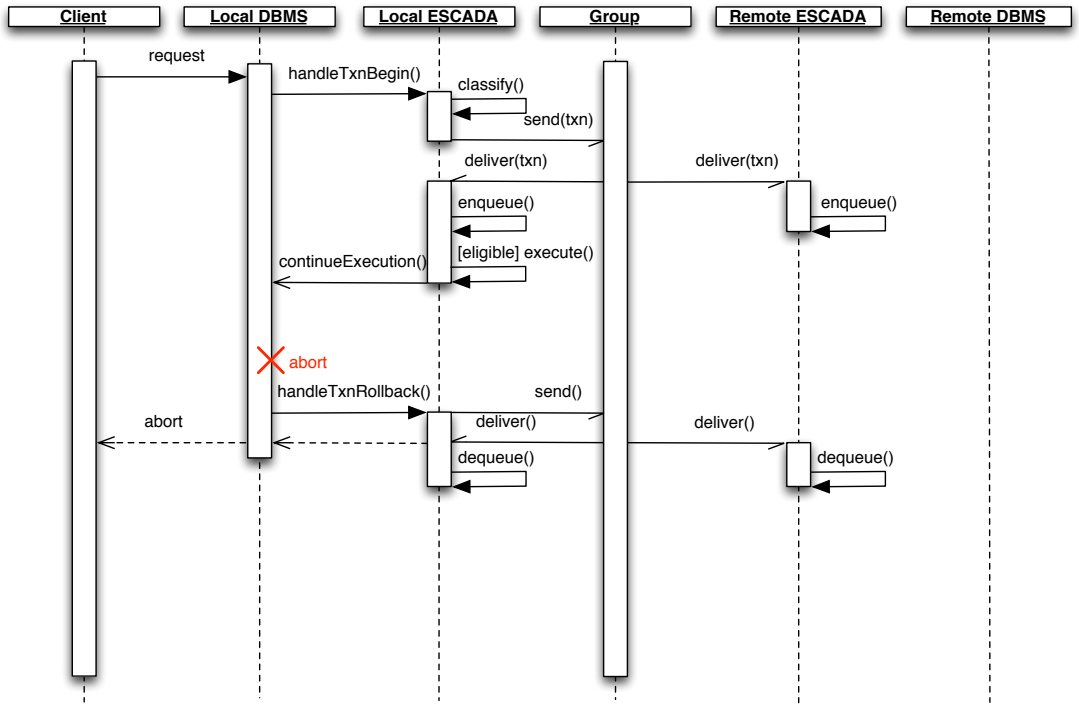


Figure 6.5: Sequence diagram for the ESCADA implementation of AJITTS: the transaction is aborted by the database's local concurrency control.

Steps 1 to 4: Same as above.

Step 5: At the local replica, the transaction is aborted by the database's concurrency control when *executing*, triggering a coordination kernel's handler invocation. The transaction moves to the *aborting* state. A notification of the transaction's abort is sent to the group.

Step 6: Upon delivery, the transaction enters the *aborted* state and is removed from the replicated queue.

In the case of the latter:

Steps 1 to 5: Same as when the transaction commits.

Step 6: When the write set is delivered, each replica independently decides the transaction cannot be certified and its state becomes *aborting*. At the originating replica, the coordination kernel notifies the database to abort the transaction. At the remote replica it enters the *aborted* state and is simply removed from the queue.

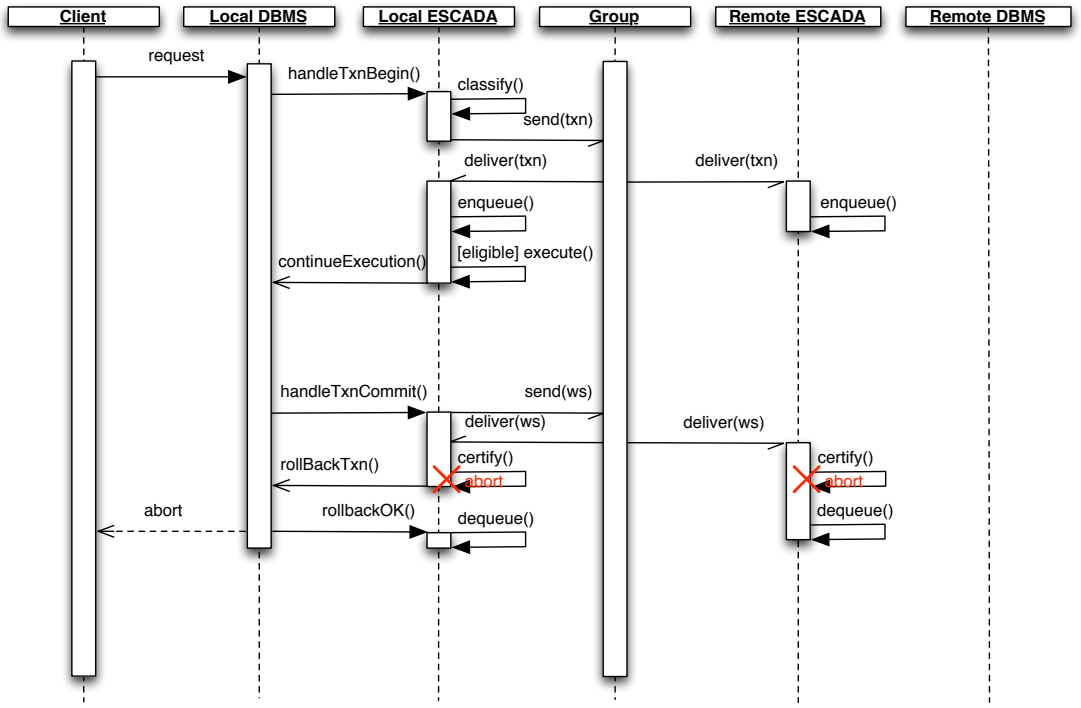


Figure 6.6: Sequence diagram for the ESCADA implementation of AJITTS: the transaction cannot be committed and must be rolled back.

Step 7: Upon confirmation of the rollback, the transaction enters the *aborted* state and the transaction is removed from the local queue.

Notice that transaction write sets are sent to the group as soon as these are known. The idea is to ensure that even for large write sets, when a remote transaction reaches the head of the queue, it can be certified waiting as little as possible (if at all) for the write set to be delivered.

6.2.1 Certification

Certification must be guaranteed to be deterministic. In order to implement a certification mechanism that is external to the database engine, it is required that once a transaction is being certified, the decision to commit or abort is completely determined by the certification mechanism, meaning that no other event should be able to cause the transaction to be aborted. In particular, in order to ensure committing transactions get all necessary locks regardless of being held by concurrent executing transactions, the GAPI requires the implementation

of high-priority (*i.e.* MASTER) transactions, which are never aborted by the database engine. Still, consider the following example: let a and b be conflicting transactions executing at a given replica such that a precedes b in the commit order. Suppose a finished executing and is at the head of the queue, ready to be certified. Meanwhile, transaction b is being executed. If the certifier decides to commit a, there can be a local race between the database's deadlock resolution mechanism [1], that can decide to abort a and the incoming notification to commit it. Suppose the abort wins the race. Because transaction information is sent (and received) before the transaction commits (or aborts), remote replicas would likely commit a, leaving the system in an inconsistent state. Notice that deferring the dissemination of the transaction's information to after the transaction has been committed would make the protocol asynchronous (lazy).

The solution is to promote transactions that have finished executing to the high-priority status before notifying ESCADA, thus ensuring that the transaction will not be aborted by the database engine, *i.e.*, transactions in the *executed* state have already been promoted to MASTER. In the example above, transaction a would have already been promoted to MASTER and the race condition would not exist.

However, the promotion mechanism means that implicit certification cannot be implemented: when attempting to commit a transaction, the database engine's concurrency control mechanism is unable to abort conflicting MASTER transactions. Explicit certification must be used instead. Also, the transaction promotion mechanism can lead to deadlocks. The issue is two-fold:

local vs remote a local *executed* transaction (*i.e.* already promoted to MASTER) may block the application of a preceding and conflicting remote transaction, causing a deadlock as the remote transaction in turn blocks certification until successfully committed;

local vs local due to out-of-order execution, local *executed* transactions may also block the execution of a preceding and conflicting local transaction t at the head of $Q(t)$, also causing a deadlock as certification is blocked until t either commits or aborts.

In order to avoid local vs remote deadlocks, local transactions that are block-

[1] In PostgresSQL, the deadlock detection mechanism is based on timeouts.

ing preceding transactions from proceeding need to be detected and aborted. This is done in two stages:

pre-certification when a local transaction t finishes executing and enters the *executed* state, if it conflicts with any transaction in the *committing* state, t is aborted;

post-certification when a transaction enters the *committing* state, if it conflicts with any transaction t in the *executed* state, t is aborted;

To illustrate why both pre and *post-certification* are necessary, consider the following scenarios. First, suppose there is no *post-certification*. Consider conflicting transactions a and b so ordered, where a is a remote transaction and b is local, and the following sequence of events: (1) b has finished executing and enters the *executed* state, but because a is not yet *committing* b is not aborted by *pre-certification*; (2) a is certified and enters *committing*; (3) deadlock. *post-certification* would have aborted b at (2) allowing a to proceed.

Now suppose there is no *pre-certification* and this sequence of events: (1) a is certified and enters *committing* and b is not aborted by *post-certification* because it is not in the *executed* state and its write set is still unknown; (2) before a is committed, b finishes executing and enters *executed*; (3) deadlock. pre-certification would have aborted b at (2), allowing a to proceed.

If both a and b are local transactions and b, whether it started to execute before a or not, reaches *executed* before a does neither *pre-certification* nor *post-certification* would abort b since a never reaches *committing*. The issue is that at the coordination kernel level, this situation is indistinguishable from a simply taking a long time to execute.

At the coordination kernel level, the transaction duration estimate can be used as a hint of whether a deadlock should be suspected: if a transaction t is not preceded by any conflicting transaction yet to be committed and if the time the transaction has been in the *executing* state is significantly longer than the estimate for its type, a deadlock is suspected and resolution ensues; if not, a timer is set according to the estimate and the transaction is allowed to continue executing until it expires, before suspecting a deadlock.

If a deadlock is suspected, the potentially conflicting transactions are those in the *executed* state. At the coordination kernel level, there is no way of knowing

which transaction is blocking t. Notice that it is possible that multiple transactions block t without blocking each other. Several policies can be defined for deadlock resolution varying between aborting all potentially conflicting transactions to aborting one at a time and waiting to see if t is able to proceed, selecting the transaction to be aborted according to some heuristic. In the current implementation of AJITTS the policy is to abort all potentially conflicting transactions. Notice that the adaptive mechanism already addresses this issue by executing transactions as late as possible without reducing throughput. Notice that in a in-database implementation of AJITTS, it would be possible to determine the offending transaction by examining the acquired locks.

ESCADA features a batch apply mechanism: certified transactions are submitted to the applier, which tries to commit as many as possible in parallel. Conflicting transactions are applied serially, but non-conflicting transactions are applied in parallel even if certification is done serially.

6.3 Workload

The DBT-5 implementation of the TPC-E benchmark was used to test AJITTS' implementation in ESCADA with the PostgreSQL database engine. DBT-5 offers two alternative implementations of TPC-E transactions as stored procedures: in PGSQL, native to PostgreSQL and C. The C version of the stored procedures was used except for the Data Maintenance transaction, which presented issues. The PGSQL version of the Data Maintenance stored procedure was used instead. The current version of ESCADA does not support composite primary keys. These were replaced with surrogate keys based on a sequence and the composite key constraint is enforced by a uniqueness constraint over an index of its components: *e.g.*, to

```
CREATE TABLE account_permission (
    ap_ca_id IDENT_T NOT NULL,
    ap_acl VARCHAR(4) NOT NULL,
    ap_tax_id VARCHAR(20) NOT NULL,
    ap_l_name VARCHAR(30) NOT NULL,
    ap_f_name VARCHAR(30) NOT NULL)
```

where `ap_ca_id, ap_tax_id` is the composite primary key of this table,

```
ALTER TABLE account_permission
ADD COLUMN ap_id SERIAL UNIQUE,
ADD CONSTRAINT pk_account_permission
PRIMARY KEY (ap_id)

CREATE UNIQUE INDEX old_pk_account_permission
ON account_permission (ap_ca_id, ap_tax_id);
```

is added.

Also ESCADA does not support foreign key constraints.

Another issue is that TPC-E defines custom data types, which were created in PostgreSQL. The necessary mappings between PostgreSQL datatypes and Java object types, needed by the reflection mechanism were added to the GAPI.

The test setup consisted of three identical replicas, each with a dual-core Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz with 8 GB of RAM and a Serial ATA, 7200 rpm disk drive, connected through a local switched network.

Each replica runs: a PostgreSQL database engine, an instance of the ESCADA framework and an instance of the DBT-5 benchmark. Each instance of the DBT-5 benchmark submits connects directly to the PostgreSQL database engine to submit transactions. Before the benchmark is run, each database is loaded with the data for 5000 customers and 30 initial trade days. Each benchmark instance runs with 1000 active customers, with 5 clients, for 30 minutes. Notice that this deployment is considerably smaller than the one simulated in Chapter 4. The key contribution in this chapter is to show that AJITTS can in fact be implemented and that it works as expected.

6.4 Results

Figure 6.7 shows how transaction latency breaks down in terms of how long it took to execute the transaction and the pre- and post- execution delays, aggregated from all replicas. Like Figure 5.1, it showcases the effect of the *input* parameter on transaction latency, as smaller values of input tend to increasingly shift latency from the post- to the pre-execution delay as on the right transactions tend to execute earlier while on the left, transactions tend to execute later. Also, some variation can be seen on transaction duration.

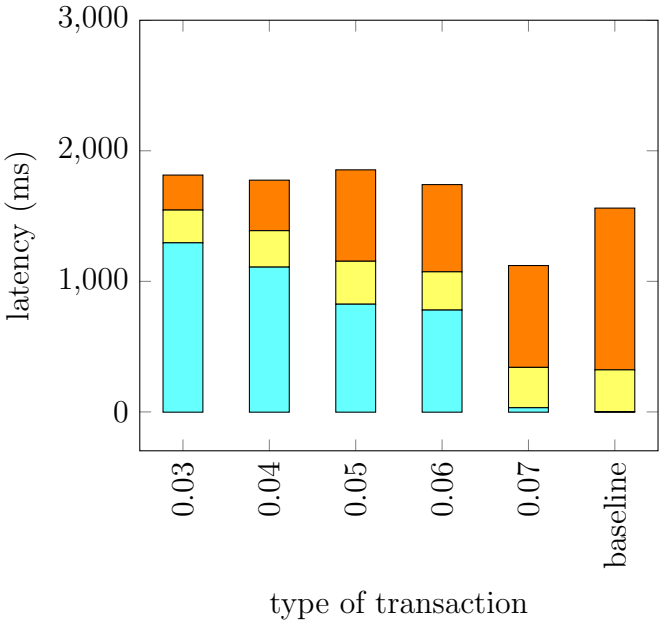


Figure 6.7: Global latency breakdown with a varying scheduler parameter: pre-execution delay (blue), execution latency (yellow), and queueing before certification (orange)

From Figures 6.8 and 6.9, while there is a small increase in overall transaction latency when comparing AJITTS using a fixed *input* to the baseline protocol, the desired effect of shifting latency from post- to pre-execution latency is clearly demonstrated.

Figure 6.10 shows how AJITTS performs with full adaptation using a setpoint value approximately equal to the mean global transaction duration. Notice that the behaviour is similar to what is shown in Figure 5.6, where transactions that, on average, take longer to execute, have, on average, larger post-execution delays. In Figure 6.11, the setpoint value approximates twice the mean global transaction duration and as shown, the system globally adapts to reach it.

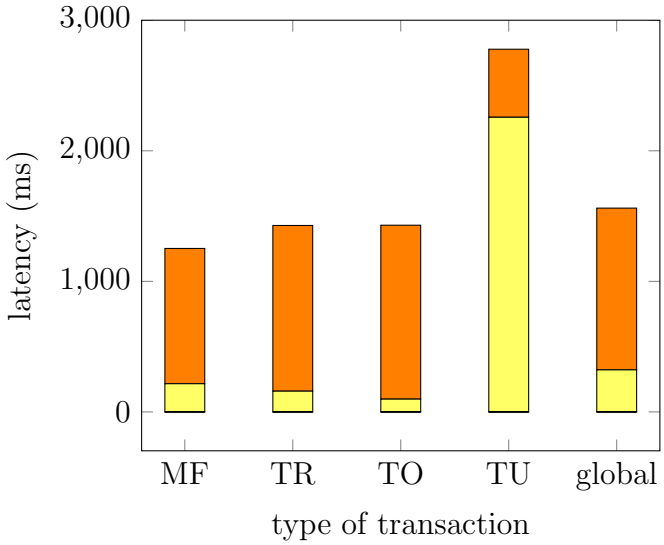


Figure 6.8: Combined transaction latency breakdown per type of transaction using the baseline protocol: pre-execution delay (blue), execution latency (yellow), and queueing before certification (orange)

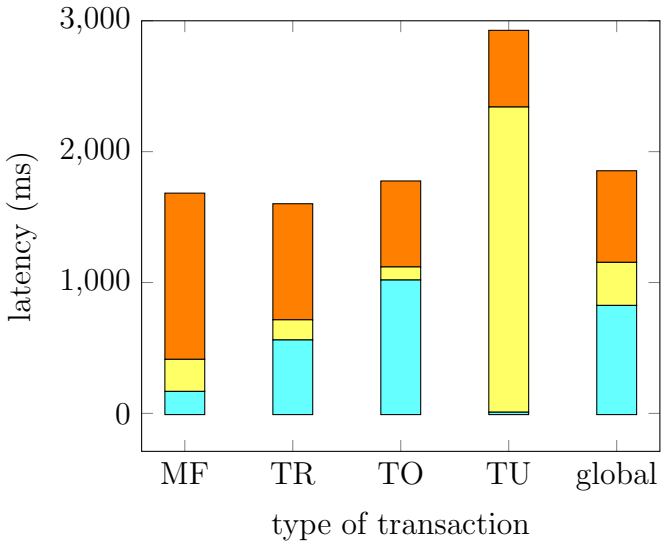


Figure 6.9: Combined transaction latency breakdown per type of transaction using AJITTS with $input = 0.05$: pre-execution delay (blue), execution latency (yellow), and queueing before certification (orange)

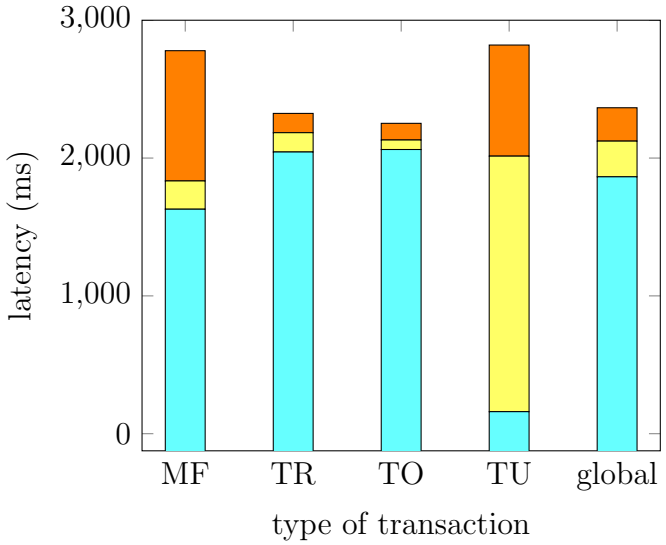


Figure 6.10: Combined transaction latency breakdown per type of transaction using AJITTS with *setpoint* = 200: pre-execution delay (blue), execution latency (yellow), and queueing before certification (orange)

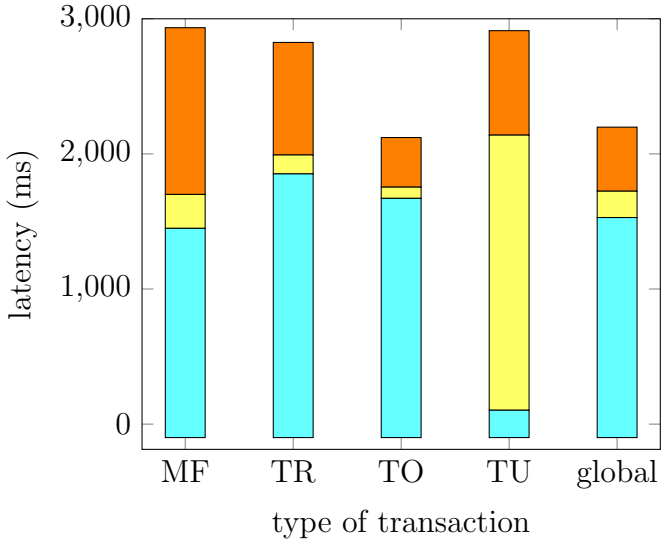


Figure 6.11: Combined transaction latency breakdown per type of transaction using AJITTS with *setpoint* = 500: pre-execution delay (blue), execution latency (yellow), and queueing before certification (orange)

6.5 Summary

This chapter described a prototype implementation of AJITTS, evaluated using the TPC-E benchmark on multiple replicas. Results demonstrated that the post-execution delay can be efficiently moved to before the transaction executes in practice, using the adaptive mechanism described in the previous chapters, which at a large scale deployment will produce the performance impact predicted by simulation.

Chapter 7

Conclusion

Replication is key to achieving highly-available dependable database management services. There is a large base of production systems that rely intrinsically on relational databases that must be able to scale to accommodate growing user bases or evolving markets. Current proposals based on key-value stores that focus on high-availability while relaxing consistency guarantees might not be suitable for a considerable subset of these, particularly if the consistency afforded by the transactional model is key to their operation.

The ability to scale a distributed system based on transactional replication is determined by the level of concurrency the system can support while guaranteeing correctness. Concurrency control is key and strategies fundamentally differ on whether conflict detection is done conservatively, *a priori*, or optimistically, after transaction execution. Both have drawbacks: contention for the conservative strategy and high abort rates when loaded for the optimistic strategy. The applicability of the conservative strategy was evaluated by assessing whether assumptions about the ability to conveniently partition application databases, critical for performance, hold for a complex benchmark such as TPC-E or a real-world application in the same domain. The analysis consisted of partitioning application databases so that a corresponding definition of disjoint conflict classes would enable the highest level of concurrency: TPC-E was analysed by manual inspection; the case-study real-world application was analysed with a tool that enabled conflict-class extraction. In both cases, the previous assumptions were found not to hold, making conservative concurrency control unsuitable.

In order to mitigate the high abort rate issue of optimistic concurrency control,

we proposed AJITTS, a transaction scheduler that minimizes the length of time during which transactions are vulnerable to being aborted. Transaction duration estimates and the level of the queueing in the system are used as the basis for the adaptive mechanism that delays transaction execution, so that it starts as late as possible, minimizing aborts, but as early as needed for throughput not to suffer.

AJITTS was evaluated in a simulated environment, using a workload based on TPC-E, and found that it outperforms the baseline protocol in which transactions are executed immediately after submission, by improving both throughput and the abort rate, particularly as the number of clients increases or the likelihood that transactions conflict rises.

A prototype implementation of AJITTS on the ESCADA framework was evaluated using TPC-E, to demonstrate that, in practice, the adaptive mechanism is able to minimize the length of time during which transactions are vulnerable to being aborted, by reaching and maintaining the selected level of queueing.

7.1 Future Work

The evaluation of AJITTS' implementation in the ESCADA framework is limited by poor scalability. It would be interesting to implement AJITTS in more recent/efficient replication frameworks, such as, *e.g.*, Galera Cluster.

Because optimistic concurrency control mechanisms are increasingly popular in cloud-based settings, it would also be interesting to implement AJITTS in cloud-based certifiers. For example, AJITTS can be implemented in OMID (Gomez Ferro et al. 2014), where transactions can be scheduled by, instead of providing a transaction with a start time stamp immediately upon request, it is delayed according to the threshold mechanism.

On a different note, the adaptive mechanism can be useful with other goals. For example, Amazon prices DynamoDB according to the contracted throughput per hour, where exceeding transactions are aborted. The adaptive mechanism can be used to keep throughput as close as possible to the contracted limit, without exceeding it, and still minimizing aborts.

Assuming partial replication and complex transactions broken up per accessed partition, AJITTS could be used to schedule each sub-transaction as late as possible to minimize aborts, but so that these finish as close in time as possible.

Bibliography

K. Aström and T. Hägglund. Automatic tuning of simple regulators with specifications on phase and amplitude margins. *Automatica*, 20(5):645 – 651, 1984. ISSN 0005-1098. doi: 10.1016/0005-1098(84)90014-1. URL `http://www.sciencedirect.com/science/article/pii/0005109884900141`. - **Cited** on page 60.

K. Aström and R. Murray. Feedback systems: An introduction for scientists and engineers. Technical report, Princeton University Press, 2007. - **Cited** on pages 56 and 59.

J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011. - **Cited** on page 3.

N Carvalho, A. Correia Jr, J. Pereira, L. Rodrigues, R. Oliveira, and S. Guedes. On the use of a reflective architecture to augment database management systems. *J. UCS*, 13(8):1110–1135, 2007. - **Cited** on page 84.

T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007. - **Cited** on page 3.

F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008. - **Cited** on page 3.

G. Chartrand and L. Lesniak. *Graphs & Digraphs*. Chapman & Hall, 1996. - **Cited** on pages 36 and 43.

A. Cheung, S. Madden, O. Arden, and A. Myers. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment*, 5(11):1471–1482, 2012. - **Cited** on page 21.

A. Correia, J. Pereira, and R. Oliveira. Akara: A flexible clustering protocol for demanding transactional workloads. *On the Move to Meaningful Internet Systems: OTM 2008*, pages 691–708, 2008. - **Cited** on pages 9, 13, 14, 16, 21, 26 and 53.

A. Correia Jr, A. Sousa, L. Soares, J. Pereira, F. Moura, and R. Oliveira. Group-based replication of on-line transaction processing servers. *Dependable Computing*, pages 245–260, 2005. - **Cited** on pages 7, 29 and 44.

A. Correia Jr, J. Pereira, L. Rodrigues, N. Carvalho, R. Vilaça, R. Oliveira, and S. Guedes. Gorda: An open architecture for database replication. In *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*, pages 287–290. IEEE, 2007. - **Cited** on pages 82, 84 and 85.

C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010. - **Cited** on pages 21, 23 and 28.

X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4): 372–421, 2004. - **Cited** on page 8.

N. Diegues and P. Romano. Bumper: Sheltering transactions from conflicts. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 185–194. IEEE, 2013. - **Cited** on page 16.

D. Gomez Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 676–687. IEEE, 2014. - **Cited** on pages 12, 49 and 100.

R. Guerraoui and A. Schiper. The generic consensus service. *Software Engineering, IEEE Transactions on*, 27(1):29–41, 2001. - **Cited** on page 8.

S. Guo, W. Sun, and M. Weiss. Solving satisfiability and implication problems in database systems. *ACM Transactions on Database Systems (TODS)*, 21(2): 270–293, 1996. - **Cited** on page 44.

S. Hirve, R. Palmieri, and B. Ravindran. Archie: a speculative replicated transactional system. In *Proceedings of the 15th International Middleware Conference*, pages 265–276. ACM, 2014. - **Cited** on page 15.

P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010. - **Cited** on page 21.

R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 477–484. IEEE, 2002. - **Cited** on pages 7, 20 and 25.

B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 134–143, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3. URL `http://dl.acm.org/citation.cfm?id=645926.671855`. - **Cited** on pages 7, 12, 14, 25, 50 and 82.

B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 424–431. IEEE, 1999. - **Cited** on pages 7, 9, 20 and 25.

T. Lahiri, V. Srihari, W. Chan, N. Macnaughton, and S. Chandrasekaran. Cache fusion: Extending shared-disk clusters with shared caches. In *VLDB*, volume 1, pages 683–686, 2001. - **Cited** on page 7.

Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM*

SIGMOD international conference on Management of data, pages 419–430. ACM, 2005. - **Cited** on pages 10, 26 and 51.

H. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proceedings of the VLDB Endowment*, 7(5):329–340, 2014. - **Cited** on page 15.

P. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 454–465. IEEE, 2011. - **Cited** on page 9.

H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 707–710. IEEE, 2001. - **Cited** on page 85.

R. Palmieri, F. Quaglia, and P. Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 59–64. IEEE, 2011. - **Cited** on page 17.

M. Patiño Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middler: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23:375–423, November 2005. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/1113574.1113576. URL `http://doi.acm.org/10.1145/1113574.1113576`. - **Cited** on page 27.

M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. *Distributed Computing*, pages 147–160, 2000. - **Cited** on pages 7 and 20.

A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72. ACM, 2012. - **Cited** on page 22.

F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*, pages 175–182. IEEE, 1997. - **Cited** on page 16.

F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14:71–98, 2003. ISSN 0926-8782. URL `http://dx.doi.org/10.1023/A:1022887812188`. 10.1023/A:1022887812188. - **Cited** on pages 7, 12, 14, 16, 25, 49 and 82.

D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation*, pages 4–6, 2010. - **Cited** on page 12.

J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 558–569, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi: 10.1145/564691.564757. URL `http://doi.acm.org/10.1145/564691.564757`. - **Cited** on page 21.

F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990. - **Cited** on page 8.

B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to determine a good multi-programming level for external scheduling. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, page 60, april 2006a. doi: 10.1109/ICDE.2006.78. - **Cited** on page 14.

B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *Proeceedings of the 3rd Symposium on Networked Systems Design and Implementation*, 2006b. - **Cited** on page 26.

D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1 –12, june 2012. doi: 10.1109/DSN.2012.6263931. - **Cited** on page 14.

M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007. - **Cited** on pages 16, 22 and 23.

A. Tatarowicz, C. Curino, E. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 102–113. IEEE, 2012. - **Cited** on page 22.

A. Thomson and D. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010. - **Cited** on page 9.

A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012. - **Cited** on page 21.

P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From a to e: analyzing tpc's oltp benchmarks: the obsolete, the ubiquitous, the unexplored. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 17–28. ACM, 2013. - **Cited** on page 23.

TPC Benchmark C - Standard Specification. Transaction Processing Performance Council (TPC), revision 5.0 edition, 2001a. - **Cited** on page 25.

TPC Benchmark W - Standard Specification. Transaction Processing Performance Council (TPC), revision 1.6 edition, August 2001b. - **Cited** on page 25.

TPC Benchmark E - Standard Specification. Transaction Processing Performance Council (TPC), revision 1.12.0 edition, June 2010. - **Cited** on pages 25 and 26.

M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 464–474. IEEE, 2000. - **Cited** on page 8.