Gargamel : accroître les performances des DBMS en parallélisant les transactions en écriture

Pierpaolo Cincilla

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Pierpaolo CINCILLA

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Gargamel: accroître les performances des DBMS en parallélisant les transactions en écriture

M. Guillaume PIERRE	Rapporteur
M. Patrick VALDURIEZ	Rapporteur
M. Marc SHAPIRO	Directeur de thèse
M. Alan FEKETE	Examinateur
Mme. Anne DOUCET	Examinateur
M. Sébastien MONNET	Encadrant de thèse

Contents

Part I – **Thesis** **1**

1 Introduction **3**
1.1 Problem Statement . . . 4
1.2 Research Proposal and Contributions . . . 6
1.3 Outline of the Thesis . . . 8

2 State of the art **9**
2.1 Introduction . . . 9
2.2 Distributed Databases . . . 10
2.2.1 Consistency Properties . . . 12
2.2.2 Replication Strategies . . . 15
2.2.3 Concurrency Control . . . 19
2.2.4 NoSQL . . . 21
2.3 Approaches to Scale-up Replicated Databases . . . 23
2.4 Conclusion . . . 25

3 Singe-Site Gargamel **27**
3.1 Introduction . . . 28
3.2 System Architecture . . . 29
3.2.1 System Model . . . 29
3.2.2 Scheduler . . . 31
3.2.3 Classifier . . . 31
3.2.4 Nodes . . . 33
3.3 Scheduling Algorithm . . . 33
3.3.1 Isolation Levels . . . 37
3.3.2 Correctness . . . 39
3.4 Conclusion . . . 41

4 Multi-Site Gargamel **45**
4.1 Introduction . . . 46
4.2 System Architecture . . . 47

4.3 Distributed Scheduling and Collision Resolution . . . 47
4.3.1 Collisions . . . 49
4.3.2 Collision and Synchronisation Protocol . . . 53
4.3.3 Determinism and Duplicate Work . . . 54
4.4 Fault Tolerance . . . 56
4.4.1 Scheduler Failure . . . 56
4.4.2 Node Failure . . . 58
4.5 Conclusion . . . 58

5 Simulation **63**
5.1 Simulation Model . . . 64
5.1.1 TPC-C . . . 64
5.1.2 TPC-E . . . 67
5.1.3 Round-Robin, Centralised-Writes and Tashkent+ Simulation . . . 68
5.2 Simulation Results . . . 69
5.2.1 Single-Site Performance . . . 72
5.2.2 Single-Site Resource Utilisation, Bounded Workers . . . 73
5.2.3 Single-Site Resource Utilisation, Unbounded Workers . . . 76
5.2.4 Multi-Site Gargamel . . . 78
5.2.5 Adaptation to The Number Of Workers . . . 86
5.3 Conclusion . . . 88

6 Gargamel Implementation **89**
6.1 TPC-C Client Implementation . . . 91
6.2 Scheduler Implementation . . . 91
6.3 Node Implementation . . . 92
6.3.1 Certification Algorithm . . . 93
6.4 Classifier Implementation . . . 96

7 Gargamel Evaluation **97**
7.1 Single Site Evaluation . . . 98
7.1.1 Benefits of the Pessimistic Approach . . . 99
7.1.2 Benefits of the Passive Replication Approach . . . 104
7.1.3 Resource Utilisation . . . 104
7.2 Multi-Site Evaluation . . . 107
7.2.1 Benefits of the Multi-Site Deployment . . . 109
7.2.2 Impact of Database Performance . . . 110
7.2.3 Impact of collisions . . . 113
7.3 Conclusion . . . 115

8 Conclusion **117**
8.1 Summary . . . 117
8.2 Perspectives . . . 119

Part II – **Résumé de la thèse** **123**

9 Résumé de la Thèse **125**
9.1 Introduction . . . 126
9.1.1 Définition du Problème . . . 127
9.1.2 Contributions : Gargamel . . . 129
9.2 Gargamel Mono-Site . . . 131
9.2.1 Architecture du Système . . . 132
9.2.2 Modèle de système . . . 133
9.2.3 Conclusion . . . 133
9.3 Gargamel Multi-Site . . . 134
9.3.1 Introduction . . . 134
9.3.2 Architecture du Système . . . 135
9.3.3 Conclusion . . . 136
9.4 Simulation . . . 137
9.5 Implementation de Gargamel . . . 139
9.6 Conclusion . . . 139

Bibliography **143**

Glossary **151**

Part I

Thesis

Chapter 1

Introduction

Contents

1.1 Problem Statement 4

1.2 Research Proposal and Contributions 6

1.3 Outline of the Thesis 8

Database (DB) systems have been used for decades to store and retrieve data. They provide applications with a simple and powerful way to reason about information. Their success is due to the ability to expose an (usually SQL[1]) interface to users and applications that masks the location and internal organization of data. Thanks to the separation of the logical definition of data from its implementation, applications do not need to know where and how data is stored. Data can be stored on disk, memory, locally or remotely, can be replicated or not, transparently to the application. Classic Database Management Systems (DBMSs) provide strong guarantees and a powerful transactional semantic that make simple for application access and manage their data.

[1]Structured Query Language (SQL) is a special-purpose programming language designed for managing data held in relational databases.

The pervasive nature of databases makes their fault tolerance (i.e., the ability to respond gracefully to a failure) and performance (in terms of throughput and response time) critical. Fault tolerance and performance are often addressed by replication which allows data to be stored by a group of machines. Database replication has the potential to improve both performance and availability, by allowing several transactions to proceed in parallel, at different replicas.

Database replication works well for read-only transactions, however it remains challenging in the presence of updates. Concurrency control is an expensive mechanism; it is also wasteful to execute conflicting transactions concurrently, since at least one must abort and restart. This well-known issue prevents DBMSes from making effective use of modern low-cost concurrent architectures such as multicores, clusters, grids and clouds.

The focus of this thesis is about how to scale up replicated databases efficiently to a potentially large number of replicas with full support for updates, without giving up consistency.

1.1 Problem Statement

Any database will have a concurrency control system to ensure transaction isolation. Concurrency control coordinates parallel transactions in order to avoid anomalies and to maintain invariants. The transaction isolation level determines the consistency level that a database provides.

The traditional criterion for the correctness of a concurrency control mechanism is *serialisability*, that means that the *execution history* (i.e., the sequence of the read/write operations performed) is equivalent to some serial history.

In distributed databases the concurrency does not only occur "inside" databases but also "among" replicas; concurrency control is then much more challenging because needs to synchronise execution both inside and among replicas. A natural correctness criterion is *1-copy-serialisability*. 1-copy-serialisability means that the execution of the

distributed database is indistinguishable from the execution of a single database, with serialisability as its isolation level.

The drawback of this strict correctness criterion (also called strong consistency) is that it competes with scalability. Another scalability bottleneck that we face in this thesis come from the fact that, using full replication (i.e., each replica contains all the data) a distributed database must synchronise with all its replicas each time a transaction performs an update.

There are several approaches to make distributed databases scalable. The concurrency control bottleneck can be alleviated by relaxing the isolation level [25, 60, 12], relaxing the transactional ACID properties [63, 23, 1, 23], parallelising reads [48, 50, 58], or by partial replication [59, 56, 8, 3, 40]. Stonebraker et al. in [63] claim that current DBMSs should be simply retired in favor of a collection of specialized engines optimised for different application areas such as text, data warehouses and stream processing. Each of those approaches comes with its advantages and disadvantages.

The above-mentioned approaches families work well for some classes of application, but not for others: relaxing the isolation level introduces anomalies that can potentially break application invariants. Giving up transactional ACID properties is bug-prone and difficult to get right for application developers. Parallelising reads works well for read-dominated workloads but does not scale to write-intensive ones. Partial replication is not practical in all workloads because cross-partition transactions are not well supported, and potentially inefficient. The development cost of specialized engines cannot be spread over a number of users as large as in general purpose engines, therefore developing specialized engines is often a less cost-effective approach than developing general-purpose engines.

Our proposal is to retain the familiar consistency guarantees provided by commercial databases and to provide strong transactional guarantees. We avoid replica synchronisation after each update by moving the concurrency control system *before* the transaction execution, at the load balancer level.

1.2 Research Proposal and Contributions

Databases often scale poorly in distributed configurations, due to the cost of the concurrency control and to resource contention.

We observe that it is more efficient to run conflicting transactions in sequence than in parallel because to run conflicting transactions concurrently cause aborts, that generate wasted work.

Our approach is to classify transactions according to their predicted conflict relations at a front-end to the replicated database. Non-conflicting transactions execute in parallel at separate replicas, ensuring high throughput; both read-only and update transactions are parallelised. Transactions that may conflict are submitted sequentially, ensuring that they never abort, thus optimising resource utilisation. This approach is flexible and lets the application choose the isolation level that better fits its needs. We discuss the isolation level of our prototype in Chapter 6, it is designed to allow the system to replicate transactions asynchronously; it does not require (costly and slow) global synchronisation. Our system, *Gargamel*, operates as a front-end to an unmodified database, obviating the cost of lock, conflicts and aborts. Our approach also improves locality: effectively, Gargamel partitions the database dynamically according to the transaction mix. This results in a better throughput, response times, and more efficient use of resources: our experiments show a considerable performance improvement in highly-contended workloads, with negligible loss otherwise.

Our current classifier is based on a static analysis of the transaction text (stored procedures). This approach is realistic, since the business logic of many applications (e.g., e-commerce sites OLTP applications) is encapsulated into a small, fixed set of parametrised transaction types, and since ad-hoc access to the database are rare [63]. Our static analysis of the TPC-C benchmark is complete, i.e., there are no false negatives: if a conflict exists it will be predicted. However, false positives may happen: a conflict may be predicted while none occurs at run time. False positives cause Garga-

mel to over-serialise transactions that do not conflict. For other types of workloads, if false negatives cannot be avoided, Gragamel should certify transactions after their execution, because some conflicting transactions can execute concurrently.

In order to lower client latency, Gargamel can be deployed in miltiple sites (i.e., datacenters). In multi-site configuration Gargamel uses a front-end for each site. A front-end synchronises with other front-end optimistically, off of the chritical path.

Our contributions are the following:

- We show how to parallelise non-conflicting transactions by augmenting a DBMS with a transaction classifier front end, and we detail the corresponding scheduling algorithm. Each replica runs sequentially, with no resource contention.

- We propose a simple prototype classifier, based on a static analysis of stored procedures.

- We have implemented a discrete event simulator that has been used to quickly evaluate the first ideas. We published simulation results for the single-site case in [20].

- We demonstrate the effectiveness of our approach with a prototype, varying a number of parameters, and comparing against a Round-Robin scheduler.

- We conclude from the evaluation that: (i) At high load, compared to Round-Robin systems, Gargamel improves latency by an order of magnitude, in the TPC-C benchmark. At low load, Gargamel provides no benefit, but the overhead it induce is negligible. (ii) The Gargamel approach requires far fewer resources, substantially reducing monetary cost, and scales much better than the Round-Robin approach.

1.3 Outline of the Thesis

The thesis proceeds as follows. Chapter 2 presents the state of the art. Chapter 3 details the single-site Gargamel approach. Additions for the multi-site case are explained on Chapter 4. Our simulator and its results are presented in Chapter 5. We describe our prototype in Chapter 6 and detail experimental results in Chapter 7. Finally, we conclude and give the perspective offered by this thesis in Chapter 8.

Chapter 2

State of the art

Contents

2.1 Introduction 9

2.2 Distributed Databases 10

2.2.1 Consistency Properties 12

2.2.2 Replication Strategies 15

2.2.3 Concurrency Control 19

2.2.4 NoSQL 21

2.3 Approaches to Scale-up Replicated Databases 23

2.4 Conclusion 25

2.1 Introduction

Relational databases have been used for decades to store and query data. The database consistency model, and its guarantees in terms of atomicity, consistency, isolation and durability (the well known ACID properties) give programmers a clear and powerful

mechanism to manage data. Lots of applications rely on databases software, making their fault tolerance and scalability critical.

The most common way to achieve fault tolerance and scalability is to replicate, i.e., copying and maintaining database objects in multiple databases, called "replicas". Moreover, the emergence of cheap commodity hardware has made replication even at (very) large scale practical.

Database replication remains challenging. To maintains the classic ACID guarantees in a replicated database, and to keep all replicas up-to-date at every point in time involves synchronising all replicas every time an update operation occurs in order to coordinate concurrent writes and the update dissemination. This creates a synchronisation bottleneck wich is an obstacle to scalability.

2.2 Distributed Databases

When a system is replicated either for performance or fault tolerance it becomes important to be able to scale to a large number of replicas, which may even be geo-distributed, i.e., spread in different geographical regions.

Replication has the potential to improve performance by allowing several transactions to proceed in parallel at different replicas [45]. Therefore, the number of replicas in a distributed database impacts the degree of parallelism it can achieve, thus the system throughput.

Replica placement (i.e., the locations where replicas are placed) plays an important role for both fault tolerance and the latency the system can provide to clients. To increase disaster-tolerance, replicas should be far away from one an other (e.g, in different datacenters). In order to reduce the client-perceived latency, we need to place replicas as close as possible to clients. This make geo-replication an attractive, but even more challenging, approach.

A replicated database needs global synchronisation to coordinate write accesses to replicated data and to disseminate updates at all replicas. Update dissemination (also called *anti-entropy*) keeps replicas up-to-date and avoids divergence.

In the case of multi-master replication (also known as multi-primary), more than one replica can update the same record, data access and update dissemination become even more challenging because of the need to handle concurrent write conflicts and concurrent update dissemination. This requires global synchronisation among all replicas. This global synchronisation creates a scalability bottleneck that get worse as the number of replicas and as communication latency increases.

This scaling issue is generally addressed by partial replication and/or by giving up consistency.

In partial replication, data is striped among replicas in such a way that no replica contains the full database. The benefit of partial replication is that only a subset of the replicas need to synchronise. This overcomes a scalability limit of the full replication approach: in full replication a part of the resources (CPU, memory, etc…) of each replica is used to install the write-sets produced by the other replica (or to re-execute *remote* transactions). Thus, while increasing the number of replicas, the system reach a point in which adding replicas does not expand its capacity anymore [57]. This works well for easily and clearly partitionable datasets, but is problematic for workloads with large or unpredictable data access patterns, because cross-partition transactions are complex and potentially inefficient [10]. The distribution of data across partitions is critical for the efficiency of the partial replication approach. Since cross-partition access is less efficient and requires more synchronisation and coordination, excessive communication among partitions can easily offset any gains made by partial replication.

Another, orthogonal, approach is to compromise in consistency to achieve better scalability. In 2000 Eric Brewer pointed out the consistency-availability relation, conjecturing that a system cannot guarantee at the same time consistency, availability and partition tolerance (the well-known CAP conjecture). Brewer's conjecture was proved

correct two years later by Gilbert and Lynch [28].

Since large systems cannot avoid partitions, the CAP theorem requires large distributed-system architects to consider the fact that, at least during partitions, they cannot provide consistency and availability. In order to build an available partition-tolerant system, the synchronisation required for write operations and update dissemination is relaxed in a number of ways affecting system consistency and transaction isolation.

Unfortunately, consistency requirements are very application-dependent and there is not a clear general way to determine which kind of consistency is necessary to maintain applications invariants. While some class of applications can greatly benefit from the degree of parallelism offered by a weakly-consistent model, others need strong guarantees in order to be correct. Our approach is to adapt to several consistency criterion while maintaining full ACID transactional semantics.

2.2.1 Consistency Properties

To discuss about database consistency we have to look a little bit closer to the ACID properties cited in the introduction.

The A.C.I.D. properties of a transaction can be summarized as follow:

Atomicity: a transaction is executed "all or nothing", meaning that its operations are either all committed, or all rollbacked.

Consistency: every transaction, when executing in isolation, brings the database from a valid state to another valid state. A valid state is a state that accomplish all database invariants.

Isolation: intermediate (non-committed) updates of a transactions can not be observed by another transaction.

Durability: once a transaction is committed, its modifications are persistent and not subject to data loss in case of power loss, errors or crashes.

The word consistency is used differently in the CAP theorem and in the database ACID properties, which can be confusing. In the CAP theorem, it refers to consistency of replicated data: at the same point on time, all nodes see the same data; whereas in the ACID properties it refers to the "valid" state of a database, i.e., the compliance of all its invariants. When discussing about consistency we refer to "serialisability": every concurrent transaction execution history is equivalent to some sequential history [32]. From this point of view, database consistency depends on Isolation rather than the Consistency (in the sense of ACID).

The traditional correctness criterion in distributed databases is one-copy-serialisability [19] as defined by Bernstein and Goodman [13]. One-copy-serialisability imposes that the behaviour of distributed database is indistinguishable from the behaviour of a single replica at the serialisability isolation level. However, because of the performance degradation due to the requirement to keep strong consistency, and in order to benefit from parallelisation, databases relax the serialisable isolation guarantee and offer a set of weaker isolation levels.

Depending on the guarantees they provide, they have different synchronisation requirements, allow different anomalies, and have different availability characteristics [9]. We discuss Snapshot Isolation (SI), one of the most popular isolation levels, and some of its descendants.

2.2.1.1 Snapshot Isolation

Berenson et al. [12] introduce SI and others isolation levels along with anomalies that they allow. SI is one of the most popular isolation level and is the default for several database products, such as Oracle, Postgres and Microsoft SQL Server [36, 30, 22, 2]. Moreover, many databases offer SI as their strongest isolation level [25].

In SI, each transaction reads from a snapshot of the committed data. Readers never block writers and vice-versa, because reads come from a consistent snapshot, not influenced by ongoing uncommitted transactions. A read-only transaction always commits, requiring no synchronisation. Fekete et al. in [26] demonstrate that under certain conditions, transactions executing under SI produce a serialisable history. SI preserves the so-called *session guarantees* [65, 68]. A session is the abstraction of read and writes performed during an application execution. The four session guarantees are:

Read Your Writes: each read reflects previous writes an the same session.

Monotonic Reads: successive reads reflect a non-decreasing set of writes.

Writes Follow Reads: a write is propagated after the reads on which it depends. Writes made during the session are ordered after any writes whose effects were seen by previous reads in the session.

Monotonic Writes: a write is propagated after the writes that logically precede it. A write operation w_1 logically precedes a write operation w_2 if w_2 follows w_1 within the session.

These guarantees are important because without of them it is difficult to reason about data, confusing users and applications.

2.2.1.2 Generalized Snapshot Isolation (GSI) And Prefix-Consistent Snapshot Isolation (PCSI)

SI was defined in [12] for single databases and it imposes transactions to read from the latest snapshot. Distributed databases need to synchronise to guarantee this requirement. This synchronisation step can potentially delay transaction execution. This delay also affects read-only transactions, and compromises one of the main benefits of SI, to never delay or abort read-only transactions.

Elnikety et al. [25] have extended the SI definition to overcome this issue. They propose GSI, an isolation level that allows transactions to read an "older" snapshot. In GSI a read-only transaction is never delayed or aborted and does not causes update transactions to delay or abort. However, update transactions are more likely to abort: the older is the snapshot in which they execute, the larger the risk of a concurrent write causing the abort. Another observation is that GSI as is defined allows snapshots to be arbitrary old, and does not impose *monotonicity* of snapshots (i.e., for any two consecutive snapshots provided by the same replica the second snapshot is at least as fresh as the frist). This violates the session guarantees. A further refinement is PCSI [25]: a transaction must read a snapshot that contains at least locally-committed transactions. PCSI maintains the desirable properties of GSI, with the guarantee that a transaction sees at least the writes that have committed at the local replica. If a client always contacts the same replica, PCSI maintains session guarantees.

2.2.1.3 Parallel Snapshot Isolation (PSI)

More recently, Sovran et al. have introduced PSI [60], an isolation level similar to SI, which allows the system to replicate transactions asynchronously. It is designed for geo-replication and does not require a global transactions ordering. The key observation is that SI imposes a global order of all snapshots. In geo-replicated systems this is expensive, because it imposes to coordinate transactions on commit, even if there are no conflicts. PSI allows different commit orderings at different sites preserving *causal ordering* [39]: if a transaction T_2 reads from T_1 then T_1 is ordered before T_2 at every site. The idea is to provide SI inside datacenters and relax SI to PSI across datacenters.

2.2.2 Replication Strategies

There are several approaches to coordinate transaction execution and update dissemination. Each approach comes with its advantages and drawbacks.

2.2.2.1 Active Replication vs Passive Replication

Two main approaches for update dissemination are: *active replication*, or state machine replication [39, 47, 54] and *passive replication* or primary-backup replication [14, 19].

In an active replication approach, all replicas execute all operations in the same order. The assumption is that transactions are *deterministic*. A transaction can be both non-deterministic itself (e.g, because of some operation based on actual timestamp) or have non-deterministic execution (due to the DBMS or the underlying operating). In the second case, achieving determinism can be hard, especially in multithreaded or distributed concurrent systems.

In passive replication, an update is performed at a single distinguished site, usually called the "primary" site, then state updates are sent to the other replicas that update their state. This overcomes the requirement for determinism because a transaction is executed once and then each replica will receive and apply the same state updates. Passive replication approaches can be further divided into single- or multi-primary. In single-primary systems, there is only one replica that may process update transactions, whereas in multi-primary systems several (or all) replicas can perform updates. These two approaches are discussed in Section 2.2.2.3.

The trade-off between the two approaches is a balance between computational resources and bandwidth usage. Active replication generally uses more computational resources than passive replication, because it executes all operations (i.e., the transaction code) at each replica. On the other hand, if the updated state is large, passive replication will be more bandwidth-greedy because it broadcasts the state updates of all transactions. Another consideration is the efficiency and the scale-up of the two approaches. In active replication, increasing the number of replicas does not improve throughput of updates because updates transactions will be executed identically at every site. In passive replication, increasing the number of replicas can improve the update throughput (up to a saturation point) if executing transactions takes longer than just applying their write-set, as is usually the case.

In general, if transactions are not deterministic, or are costly to execute and produce a relatively small write-set (which is the case for the TPC-C benchmark, as we show in the evaluation chapter) passive replication will scale better. If, conversely, transactions are relatively short and modify lot of records, or if bandwidth is an issue, an active replication approach may perform better.

We use a passive replication approach in order to scale the update workload. In TPC-C, applying the write-set is 4 to 7 times faster than actually executing the transaction. This gives a clear scalability advantage to the passive replication schema.

2.2.2.2 Eager Replication vs Lazy Replication

Another critical factor to consider for execution latency in the update dissemination strategy, is whether replicas are updated as part of the transaction (i.e, synchronously), or asynchronously, off the critical path. Those strategies are known as *eager replication* and *lazy replication* respectively.

Eager replication protocols update all replicas as part of the transaction commit. When a transaction commits all replicas have the same value. Eager replication has the advantage that all replicas are up-to date at transaction commit: they are strongly consistent, and they have no anomalies. The drawback is that any multi-primary (also called update-anywhere) eager replication approach requires distributed locking and two phase commit [37]. Furthermore, Under eager replication, the probability of deadlock is proportional to the third power of the number of replicas [29].

Lazy replication protocols update replicas asynchronously, after the transaction commits. This improves efficiency and scalability, but gives up on consistency. Since updates are propagated after transaction commit, at a given time, replicas may have different values. This inconsistency can confuse users and applications because they are exposed to anomalies.

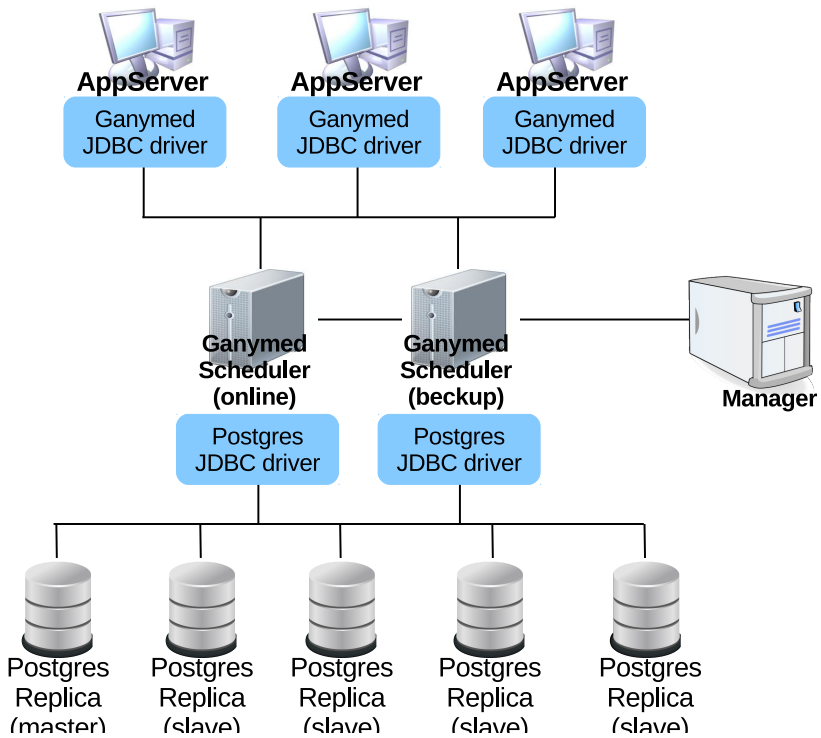


Figure 2.1: Ganymed architecture

2.2.2.3 Primary-Copy Replication vs Update-Anywhere

In a *primary-copy* approach, only one site receives all the updates. This site is called the *primary* site. It is her responsibility to broadcas updates to all the other replicas. This strategy greatly simplifies the concurrency control mechanism because the primary serialises updates.

Conversely, the *update anywhere* (also called multi-master or multi-primary) approach allows any replica to perform updates. Multi-master support parallel execution of update transactions at several replicas, however the concurrency control is much more challenging in this case, as we discute in the next section.

2.2.3 Concurrency Control

A database concurrency control mechanism is in charge of the correctness of execution of concurrent transactions. It ensures that conflicting concurrent transactions will not all commit. As discussed in Section 2.2.1 the definition of conflict depends on the targeted isolation level. Often, concurrency control ensures isolation by checking that concurrent conflicting transactions will not all commit: one or more transactions need to abort and restart. This check is done after transactions execution and is called *certification*. To reduce aborts a system can delay the execution of a transaction until every concurrent conflicting transaction has completed. Both approaches require a synchronisation step inside replicas (if they run transactions concurrently), and across replicas when the database is replicated. This synchronisation can dramatically degrade performance, especially if the number of replicas is large or communication latency is high, as in geo-replicated databases. The two main techniques used to synchronise distributed DBs are *distributed locking* and *group communication*.

With distributed locking, a transaction takes a lock on any records it will access in order to prevent conflicts. Systems usually distinguish between read- and write- locks, authorizing multiple concurrent read locks and giving exclusive access to write locks. Locking mechanisms can be either *pessimistic* or *optimistic*. In pessimistic concurrency control, locks are granted or refused when they are requested. In optimistic concurrency control, a lock is effectively taken only at the end of the transaction just before the associated record is updated. Optimistic concurrency control was proposed by H.T. Kung [38].

Schiper and Raynal point out similarities between group communication and transactions [52]. They concerns in particular the properties of atomicity and ordering. As described in Section 2.2.1, a transaction's operations must either all became visible to other transactions, or none. This is similar to the group-communication all-or-none delivery property. Ordering properties are present in both systems: a transactional system uses ordering in relation to isolation, whereas a group communication system has de-

livery order properties (e.g., First In, First Out (FIFO) that ensures that messages are received at the destination reliably and in the same order in which they was sent at the origin or Atomic Broadcast (ABCAST) that ensures that messages are received reliably and in the same order by all participants).

Our prototype uses group communication because make easy to implement various concurrency control and update propagation strategies. Our concurrency control and update propagation are based on the JGroup [42] FIFO and ABCAST primitives. See the implementation Chapter for details.

In both cases, lock-based or group communication-based, concurrency control involves synchronising all replicas. This makes it an expensive mechanism. Running conflicting transactions concurrently is wasteful, since at least one of them must abort and restart. The waste of computational resources to execute transactions that will abort, and of bandwidth to coordinate the execution, stops DBMSs from making effective use of modern low-cost concurrent architectures such as multicores, clusters, grids and clouds.

One approach to circumvent this problem is to use primary-copy (see Section 2.2.2.3). In primary-copy all updates are centralised at a single replica and the concurrency control does not need to be distributed. It is much less likely to became a bottleneck, because All updates are serialised at the primary. This is the case of both Ganymed [48] (in a cluster) and Multimed [50] (on a multicore machine).

Ganymed uses a master replica and a set of slave replicas. The Ganymed scheduler redirects each client request to one of the replicas (see Figure 2.1), all update transactions are redirected to the master replica, read-only transactions are redirected to a slave replica. Ganymed provides SI consistency: read-only transactions always see the latest snapshot. Slaves apply the write-sets in FIFO order.

Similarly, Multimed relies on a primary-copy replication approach inside a multicore machine. The database is deployed over the cores as a set of replicas, coordinated by a middleware layer that manages consistency, load balancing, and query routing. Like Ganymed, it supports Snapshot Isolation consistency, by ensuring that updates are

transmitted via total order, and by executing a transaction at a replica that has a fresh version of the accessed data.

At a larger scale, Facebook takes a similar approach: read operations are spread across multiple geographically-distant datacenters, whereas all write operations occur in a single data centre [58]. Facebook uses a three tier architecture including a web server, a *memcache* server, and a MySQL database. When an update occurs, it is first done in the primary replica located in California, causing the invalidation of the modified records in the Californian memcache. Then, the update is propagated to the other datacenters, and when the backup replicas are up-to-date their memcache is invalidated as well. This asynchronous replication allows Facebook to never delay reads from backup replicas as Ganymed and Multimed do, but implies to give up consistency: different replicas may return different values.

The conservative approach of primary-copy can result in poor response times, as it does not parallelise non-conflicting transactions. Our simulations confirm this intuition.

2.2.4 NoSQL

An alternative to classical DBMS is to give up on strong transactional consistency [68]. A new class of datastore has emerged, with the property that they do not provide a full ACID transactional semantics. They are often referred as Not Only SQL (NoSQL) data stores, but there is no general agreement on this definition [17].

According to Stonebraker [61], NoSQL data stores can be classified into *document-style stores*, in which a database record consists in a collection of key-value pairs plus a payload; and *key-value stores* whose records consist in key-payload pairs. Both do not support a full SQL interface and they have a weaker concurrency model than ACID transactions. Examples of this class of systems include CouchDB [6], MongoDB [49] (document-style stores), BigTable [18], MemcacheDB [1] and Amazon's Dynamo [23] (key-value stores). Other systems, as CloudTPS [69], Megastore [11], G-Strore [21] and

Scalaris [55] present an hybrid model that provide ACID transactional semantics on top of key-value stores.

A MongoDB record is a JSON document, which is a data structure composed of field and value pairs. MongoDB uses primary-copy replication, with asynchronous updates propagation. It lets the application choose to read from the primary (strong consistency) or from slaves (better scalability).

CouchDB shares the same record structure as MongoDB: records are JSON documents. However, in contrast to MongoDB, CouchDB uses a multi-master replication. In CouchDB, any replica can be updated, and updates are incrementally sent to other replicas. Conflicts are resolved automatically by choosing deterministically one of the conflicting versions. The discarded versions remain available to applications, so reconciliation can be handled at the application level.

Bigtable uses multidimensional distributed sorted maps indexed by a row key, a column key and a timestamp. The timestamp indexes the "version" of the row. Values are uninterpreted arrays of bytes. It uses a primary-copy replication schema and relies on Chubby [15] to ensure that there is at most one active master at a time. Updates are propagated asynchronously. Bigtable offers single-row transactions, but it does not support general transactions across rows.

MemcacheDB is a distributed key-value storage system. It uses Berkeley DB [43] as a storing backend. It uses a primary copy replication schema with asynchronous update replication. It supports transactions through Berkeley DB.

Dynamo is a multi version key-value store that uses a multi-master replication schema. Its consistency protocol is similar to those used in quorum systems: it has two configurable parameters W and R representing the minimum number of replicas that have to acknowledge the commit respectively of a write and a read operation. When the application performs a write or a read it contacts a replica that will be the coordinator for that operation. The coordinator forwards the operation to all the replicas and, if the operation is a write, it waits for the reception of the acknowledge that the write

was committed by W replicas before returning, and if the operation is a read, it waits to receive (R) replies and sends the freshest to the application. When $W + R$ is larger than the number of replicas, applications are guaranteed to receive the latest version (they can receive more than one version in case of conflicts), otherwise they can receive a stale data. Dynamo uses vector clocks [39] in order to capture causality between different versions of the same object. When conflicts are detected (thanks to their vector clocks) they are exposed to the client application that performs its own reconciliation logic (called *business-logic specific reconciliation*) or it performs an automated "last writer wins" reconciliation policy (called *timestamp based reconciliation*).

In conclusion, the NoSQL approach is promising for some particular classes of application, in particular for update-intensive and lookup-intensive workloads such as online transaction processing (OLTP). It generally needs less synchronisation than the classic database approach, but the weak consistency model makes it more bug-prone and difficult to get right for application developers.

The ideology of the NoSQL community, "Use the right tool for the job" [31] emphasizes the idea that not all applications need full transactional ACID semantics and strong consistency, and these will achieve a better scalability by embracing NoSQL.

In the other hand, "one size does not fit all" [62] and full transactional ACID semantics remain a powerful tool to manage data and make applications robust.

2.3 Approaches to Scale-up Replicated Databases

In this section we overview some systems that use the replication strategies discussed earlier (e.g., primary-copy, update-anywhere, lazy replication, active and passive replication). We also discuss how they are related to our system.

H-Store [63] is a DBMS designed and optimised for OLTP applications. It requires the complete workload to be specified in advance as statically defined *stored procedures*. This advance knowledge allows H-Store to partition and to parallelise the load between

different single-threaded replicas operating in a share-nothing environment. Under this assumption, H-Store improves the performance by orders of magnitude compared to other commercial database. Gargamel also parallelise the load between single-threaded replicas, but using an unmodified DBMS. Furthermore, Gargamel requires only an approximate knowledge, encapsulated in the conflict classifier, and it does not require the whole workload to be specified in advance.

The system of Pacitti et al.[46] is a lazy multi-master replicated database system. It enforces a total order of transactions by using reliable multicast and a timestamp-based message ordering mechanism. Their system avoids conflicts at the expense of a forced waiting time for transactions, and rely in a fast cluster network to reduce the waiting time. Gargamel, in contrast, is designed for geo-replication. In order to scale in Wide Area Network (WAN), Gargamel enforces a partial order of transactions and synchronise optimistically among distant sites.

Ganymed [48] and Multimed [50] centralize updates. In contrast, Gargamel is capable of parallelising at least some update transactions. Therefore, it does not need a master replica, which constitutes a scalability bottleneck.

Sarr et al. [51] introduce a solution for transaction routing in a grid. Their system, like Gargamel, is conflict-aware. However, they check for conflicts only in order to propagate updates among replicas in a consistent way; they do not serialise conflicting transactions, as Gargamel does.

A conflict-aware scheduling system is proposed by Amza et al. [4, 5]. Their system ensures 1-Copy Serialisability (1-Copy SER) by executing all update transactions in all replicas in a total order. Gragamel parallelises non-conflicting write transactions and transmits the write-sets off the critical path. Moreover Gargamel executes a given transaction only once, at a single replica, which ensures that replicas do not diverge in the presence of non-determinism.

2.4 Conclusion

Replicated databases improve performance and availability by parallelising transaction execution. When transaction execute in parallel, the execution correctness rely on a concurrency control mechanism that avoids committing concurrent conflicting transactions. The concurrency control synchronises the transaction execution among replicas and guarantee that, if two conflicting transactions are executed in parallel, at least one of them will abort. This is an expensive mechanism and it is also wasteful to abort transactions.

Making the concurrency control scalable is an hard task. One approach to circumvent the problem is to centralise all write operations using a primary-copy replication schema. primary-copy does not need distributed concurrency control, because only one replica can perform updates. The drawback is that it cannot parallelise update transactions, so it scales only on the read workload.

Another approach is to give up on consistency. A class of datastores, called NoSQL, achieves scalability relaxing transactional ACID semantic. This works well only for applications that do not need strong consistency and ACID operations. For application developers, the relaxed consistency model and the lack of ACID transactions, make NoSQL datastores more difficult to understand than traditional DBMSs.

Our approach is to overcome the main drawbacks of previous systems using a multi-master replication schema that supports transactional ACID semantic. We make our system scalable avoiding certification and aborts through a new concurrency control mechanism. Our concurrency control predicts conflicts *before* transaction execution in order to parallelise non-conflicting transactions and serialise conflicting one.

Chapter 3

Singe-Site Gargamel

Contents

3.1 Introduction 28

3.2 System Architecture 29

3.2.1 System Model 29

3.2.2 Scheduler 31

3.2.3 Classifier 31

3.2.4 Nodes 33

3.3 Scheduling Algorithm 33

3.3.1 Isolation Levels 37

3.3.2 Correctness 39

3.4 Conclusion 41

3.1 Introduction

Coordinating the concurrent access to a distributed database is challenging. Part of the complexity comes from the concurrency control mechanism. As discussed in Chapter 2 there are several techniques (e.g., locks and group communications primitives) to maintain replica consistency. Their complexity depends on the number of replicas.

In the lifetime of a transaction we can distinguish an execution phase and a certification phase. The execution phase is the time spent by the DB to execute the transaction code. The certification phase is the time spent by the concurrency control system to check whether the transaction can commit or should abort.

The certification phase is crucial for correctness. It can not be avoided in case of update transactions. It is generally performed after the transaction execution, to decide if the system should commit or abort the transaction (although some systems can preemptively abort transactions during the execution phase). This mechanism brings a scaling issue: it needs to be coordinated between all replicas in case of full replication and between all replicas storing the accessed data in case of partial replication. This causes a synchronisation bottleneck, which worsens with the number of replicas and communication latency.

Our key observation is that, to be optimal in terms of throughput and resource utilisation, we need to:

I. never run conflicting transactions concurrently because they will eventually abort, wasting the resources they use to run (resource optimality);

II. run transactions that do not conflict in parallel (throughput optimality).

To achieve these goals, we use a conflict estimation mechanism, to do the concurrency control pessimistically, ahead of transaction execution. Based on the conflict estimation, we can execute conflicting transactions sequentially at a same replica, and spread non-conflicting transactions to multiple replicas to execute in parallel [20]. If

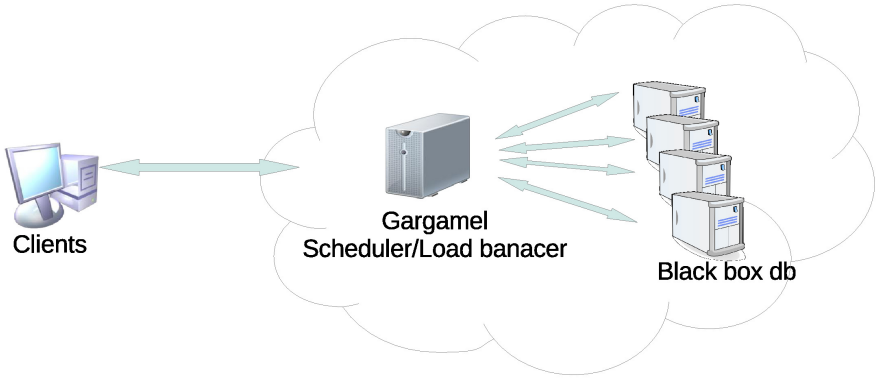


Figure 3.1: Single-site system architecture

the estimation is complete (i.e., no false positives or false negatives) then the execution will be optimal in terms of throughput and resource utilisation. Moreover, unless false negatives are possible, we do not need to certify transactions after execution, because we know they will not conflict with any concurrent transaction. Once the transaction is executed its write-set is propagated to all replicas reliably and in total order.

3.2 System Architecture

Classically, distributed DB architectures are composed of clients, which send requests to a front-end server, which forwards them to a set of nodes. Each node contains a full replica of the database . Gargamel is based on this architecture. It is located on front end, and does not require any modification to the underlying database.

3.2.1 System Model

We call the front end *Gargamel Scheduler* (or simply *Scheduler*) and we call *Nodes* replicas storing the database. Each node has one ore more processes accessing the database (usually one per CPU). We refer to those processes as *Workers* (see Figure 9.1). The database

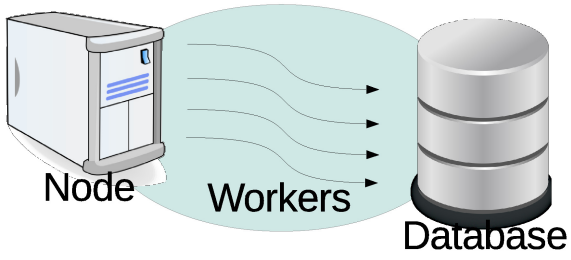


Figure 3.2: Node and Workers architecture

is fully replicated at each node.

Clients perform non-interactive transactions, they send the entire transaction "at once" (e.g., by calling a stored procedure). This restriction forbids client to abort a transaction after it is submitted. This limitation is reasonable, since the business logic of many applications (e.g., e-commerce site OLTP applications) is encapsulated into a small fixed set of parametrised transaction types and ad-hoc access to the database is rare [63].

Schedulers, nodes and clients communicate by message passing. The underlying group communication system support FIFO and ABCAST message delivery order. Communication between clients and the scheduler, between the scheduler and the nodes and between nodes and clients are done with FIFO channels. Communication between nodes use ABCAST channel to maintains a total order of transaction write-set propagation (see Section 3.3.2 and Chapter 6 for details).

The component that predicts possible conflicts between transactions is called the *transaction classifier*. Our current classifier is based on a static analysis of the transaction text (stored procedures). This approach is realistic for applications that access the database through stored procedures.

3.2.2 Scheduler

The Scheduler is the front end between clients, which submit transactions to the system, and nodes, which execute them in the database. To avoid executing conflicting transactions concurrently, the scheduler must track all conflicts detected by the classifier. Transactions and their conflict relations are stored on a directed graph, where transactions are the vertexes and conflicts are the edges (see Section 3.3 for a detailed description of the scheduling algorithm). If two transactions are linked by an edge in the graph, they are in conflict and must be serialised. Otherwise, they may be executed in parallel. The direction of the edge represents which transaction was scheduled first, and determines the execution order. If two transactions t' and t'' are connected by an edge from t' to t'' then t' will be executed before t''. We call t' a *back dependency* of t'' and t'' a *front dependency* of t'. The scheduler uses this partial order relation to parallelise non-conflicting transactions, spreading them to different nodes, and to serialise conflicting ones at a same node. Transactions are sent to nodes, along with their back dependency, to be sure that the nodes will execute them in the correct order, when their back dependency are executed remotely (see Section 3.3.2 for further details).

Once a transaction has been executed by a node and its modifications has been propagated to all other replicas, we can delete it from the graph. Indeed, since the writes of a transaction t are reflected at a node, all subsequent transactions executed at that node will be serialised after t.

3.2.3 Classifier

The classifier is in charge of determining whether or not two transactions will conflict.

Our classifier implementation is based on a very simple static analysis: two transactions are considered to conflict: (i) if they update the same column of a table, (ii) unless it is clear from the analysis that they never update the same row.

The definition of conflict is determined by the isolation level. Our current classifier

supports SI [12] and its descendants [25, 60], i.e., transactions conflict if and only if their write-sets intersect. If the system should provide another isolation level, the classifier rules may have to change accordingly. E.g., to provide serialisability, instead of checking if two transactions *write* to the same column of some table, we would check if they access the same column and at least one of the access is an update. In Section 3.2.4, we discuss how the anti-entropy mechanism impacts the isolation level.

Depending on the workload and the database architecture a classifier may have (i) *false positives*: if it considers transactions to be conflicting where in fact they do not conflict at runtime; (ii) *false negatives* if it considers as non-conflicting transactions that will indeed conflict at runtime. It is said *complete* if it does not have any false positives nor false negatives.

If the classifier is complete, then the scheduling algorithm is optimal (i) in terms of throughput, because it parallelises whenever possible but also (ii) in terms of resource utilisation, because it never runs conflicting transactions concurrently, ensuring that they never abort.

In the case of false positives, the system will serialise more than needed: some transactions will be unnecessarily executed at the same replica due to prediction of a conflict that does not effectively appear at runtime. In this case, the system is not optimal in terms of throughput. In case of false negatives it could happen that Gargamel would run conflicting transactions concurrently; at least one of them aborts, wasting resources. Moreover, in the cases where the classifier can produce false negatives, the Gargamel approach would be less advantageous because it can not avoid the certification phase, and correctness must rely on classical optimistic concurrency control. Generally, false negatives can be avoided conservatively augmenting the false positives. This is actually the case of our prototype (see Section 6.4 of Chapter 6)

3.2.4 Nodes

A node receives transactions from the scheduler and executes them sequentially. It also receives and applies write-sets of transactions executed by other nodes. Before executing a transaction, a node checks if all its back dependency are satisfied (i.e., locally committed). If not, the node postpones the execution until their write-set has been received and applied.

Once the transaction has been executed, the node acknowledges the client and, if the transaction has modified the database, it broadcasts the transaction's write-set to all the other nodes, in order to reflect the transaction update at all replicas. Replicas do not re-execute remote transactions transactions, but instead they apply their write-set, for two main reasons:

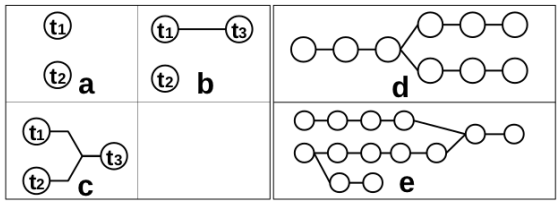
Scale update transaction. If every replica executes every update transaction, increasing the number of replicas does not improve the update throughput. Throughput will scale if applying the write-set of a transaction is faster than executing it.

Handle non-determinism. If the transaction is internally non deterministic or suffers external non-determinism (e.g., because the operating system is non deterministic), re-executing transactions at all nodes can cause database replicas to diverge. Propagating the write-set eliminates this issue.

The propagation of the write-sets among replicas impacts the isolation level properties and correctness. This is discussed later in this Chapter.

3.3 Scheduling Algorithm

As transactions enter and exit the system, conflict relations appear and disappear in complex ways. To keep track of those complex conflict relations, Gargamel maintains generalized queues, called *chains*. A chain is a sequence of transactions conflicting one



An initially empty system receives t_1, t_2, etc.: *(a)* t_1 and t_2 do not conflict. *(b)* t_1 and t_2 do not conflict, and t_3 conflicts with t_1 but not with t_2. *(c)* t_3 conflicts with both t_1 and t_2. *(d)* Split. *(e)* Split and merge.

Figure 3.3: Example scheduling scenario.

with another. Each chain is executed sequentially at a single worker. Non-conflicting transactions are assigned to parallel chains, which are executed, without mutual synchronisation, at different workers. The scheduler maintains a graph of all the transactions received grouped in chains that split and merge according to transaction's conflict relations. This graph establishes a partial order over transactions. When an incoming transaction has a conflict, it is scheduled in a chain and is sent to the worker executing that chain. If the transaction does not conflict with any other, it starts a new chain and is sent for execution to a free worker, if any. If there is any free worker the scheduler send the new chain to the less loaded worker. Therefore, when the number of workers is smaller than the number of chains, a worker executes, sequentially, more than one chain. Workers only know transactions of the chain they are executing.

Chains are built at runtime by comparing an incoming transaction with the ones that are already scheduled. The incoming transaction will be added in a chain containing all transactions that conflict with it, so that its execution will be serialised after the execution of its conflicting transactions: for some incoming transaction t, if t is classified as conflicting with t', then t is queued after t'. If t conflicts with transactions t' and t'' that are in two distinct chains, then the transaction is scheduled in a chain *merging* the two queues. Similarly, if two transactions t' and t'' both conflict with transaction t, but t' and t'' do not conflict with each other, then they will be put into two distinct queues, both

after t. We call this case a *split*.

Figure 3.3 presents some examples. Clients submit transactions t_1, t_2, etc. Initially, the system is empty: t_1 does not conflict with any transaction; therefore the scheduler allocates its first chain, containing only t_1. When the scheduler receives t_2, it compares it with the only other transaction, t_1. If they conflict, the scheduler will append t_2 at the end of the queue containing t_1; otherwise, the scheduler assigns t_2 to a new queue (Figure 3.3(*a*)).

Now consider that transaction t_3 arrives; the scheduler compares it to t_1 and t_2. If t_3 conflicts with neither of them, it is placed into a new queue. If t_3 conflicts with a single one, it is queued after it (Figure 3.3(*b*)). If it conflicts with both, the two queues are spliced into a single chain, where t_3 is ordered after both existing queues (Figure 3.3(*c*)). If transactions t_4 and t_5 both conflict with t_3 but not with each other, they will be on parallel queues, but both ordered after t_3 (the chain splits as in Figure 3.3(*c*). Repeating the algorithm, Gargamel computes chains that extend, merge or split according to conflicts between pairs of transactions (Figure 3.3(*e*)).

The number of classifications is, in the worst case, equal to the number of transactions queued and not yet committed at all replicas. Depending on the workload, a number of optimisations are possible. For example, if two transaction types never conflict, there is no need to compare transactions of one type with transactions of the other. Another case is the following: if a transaction does not conflict with some other transaction, then it will not conflict with any of its predecessors in the chain. Therefore we need to check only for the heads of each queue. In Section 6.2 we discuss how our scheduler optimises the number of classifications for the two most frequent transactions of the TPC-C workload.

Algorithm 1 describes the scheduling protocol in pseudocode. Logically, it can be divided into three parts: $scheduleLocal()$, called when a client sends a SCHEDULE message to its local scheduler, $calculateBackDependencies()$, a function that calculates conflicts between transactions in the local graph and the incoming transaction, and

$getExecutingWorker()$, which returns the worker that will execute the transaction.

The variables of the scheduling algorithm are:

- $backDependencies$*:* represents a set of all the conflicting transactions (back dependency in the graph). When a transaction is scheduled and the classifier finds a conflict with an already scheduled transaction, the already scheduled transaction is inserted in the $backDependencies$ set of the incoming transaction.
- $transactionQueue$*:* a queue used for breadth-first exploration of the graph.
- $graphHeads$*:* an hash set containing all the "heads" of the graph, i.e., transactions without any front dependency. This set contains the last scheduled transaction for each chain.
- $targetWorker$*:* the id of the worker chosen to execute the transaction.
- $precedingWorker$*:* a temporary variable used to iterate over worker candidates to execute the incoming transaction.

We now explain the algorithm's three main functions in detail:

- $scheduleLocal()$*:* is called when a client sends a transaction T_i to the Scheduler. The scheduler first computes T_i's back dependency (Line 6) and the executing worker (Line 7), then sends the transaction, along with its dependencies, to the worker for execution (Line 8).
- $calculateBackDependencies(T_i)$*:* this function takes a transaction T_i as input, and checks for conflicts with all other transactions in the graph. The checks are performed in a breadth-first exploration of the graph, starting from the last scheduled transactions (called heads). The exploration of each branch of the graph stops when: (i) a conflict is found or, (ii) the end of the branch is reached. The complexity of this algorithm is O(n), where n is the number of active transactions in the system. Optimisations are discussed in Section 6.2.

- *getExecutingWorker():* this function calculates which worker should execute the transaction. The enforced policy is to associate each chain to a worker, in order to execute conflicting transactions sequentially and to allocate separate chains to separate workers. To this end, the function distinguishes if the transaction is free (without back dependency) or if it is part of a chain (it has some back dependency). In the latter cases Gargamel checks if there exists a back dependency that "close the chain" (Lines 21–25) i.e., that has not some front dependency scheduled at the same worker (this can be the case in presence of splits). This check is necessary in order to not associate all chains following a split to the same worker.

3.3.1 Isolation Levels

As sketched in Sections 3.2.3 and 3.2.4, the isolation level property depends on both the classifier and the write-set propagation strategy.

The classifier determines when two transactions are in conflict and ensures a priori that conflicting transactions do not execute concurrently. The write-set propagation strategy determines when transaction updates became visible in the database.

In this discussion, we focus on the familiar consistency guarantees of SI and on the more recently introduced PSI. Gargamel supports the consistency guarantees of SI and PSI rather than the stronger guarantees of 1-Copy SER because they offers an higher degree of parallelisation.

SI is a common isolation level used by several commercial DBMSes (e.g., Oracle and SQLServer). It ensures that transactions are executed in a snapshot that reflects a totally ordered commit history, and that if two transactions have intersecting write-sets, then one of them will abort.

More formally SI is specified through the following properties:

SI PROPERTY 1. (Snapshot Read) All operations read the most recent committed version as of the time when the transaction began.

SI PROPERTY 2. (No Write-Write Conflicts) The write-sets of each pair of committed concurrent transactions are disjoint.

This isolation level is not well suited for distributed databases [60] because it imposes a total order of the transactions, even those that do not conflict. This total order implies a synchronisation among all replicas in the commitment phase.

PSI was introduced to overcome this issue [60]. The key difference between SI and PSI is that PSI does not impose a total order on all transactions, but only conflicting ones. Thus PSI allows asynchronous write-set propagation. Despite this it preserves two important properties of SI: i) committed transactions do not have write-write conflicts and ii) *causal ordering*: if a transaction T_2 reads from T_1 then T_1 is ordered before T_2 at every site. PSI can be appropriate for web applications, where users expects to see the effects of their own actions immediately and in order, but can tolerate a small delay for their actions to be seen by other users.

PSI is specified replacing the properties of SI with the following three properties:

PSI PROPERTY 1. (Site Snapshot Read) All operations read the most recent committed version at the transaction's site as of the time the transaction began.

PSI PROPERTY 2. (No Write-Write Conflicts) The write-sets of each pair of committed somewhere-concurrent transactions must be disjoint.

PSI PROPERTY 3. (Commit Causality Across Sites) If a transaction T_1 commits at a site A before a transaction T_2 starts at site A, then T_1 cannot commit after T_2 at any site.

In Gargamel, since both SI and PSI share the same definition of conflict, both are enforced using the same classifier policy: two transactions conflict if their write-set intersects.

The efficiency advantage of PSI derives from its write-set propagation strategy. Whereas SI imposes a total order on write-set propagation, PSI disseminates the write-set asynchronously (delivered in causal order) and commits right away.

The Gargamel prototype currently supports both SI and PSI. In the future it can easily be extended to provide 1-Copy SER.

3.3.2 Correctness

Gargamel supports the SI isolation level thanks to the following properties:

(i) It ensures that conflicting transactions execute sequentially. This implies *a fortiori* that a transaction commits only if its updates do not conflict with a concurrent transaction.

(ii) Workers propagate their updates using atomic broadcast. All replicas receive all the write-sets in the same order.

(iii) Each replica's database engine ensures SI locally. A worker does not start executing a transaction until it has received the updates of conflicting transactions.

(iv) A worker does not commit an update transaction until its write-set has been propagated at all other replicas.

To explain property *(iii)* in more detail, note that, since communication between the scheduler and replicas is asynchronous, a replica might receive an update transaction from the scheduler before being ready. When the scheduler sends a new transaction to some replica, it piggy-backs the list of back dependency. The replica checks that the corresponding updates have been received; if not, the new transaction is delayed. Property *(iv)* ensures that all transactions became visible at all sites in the same order.

Properties *(iii)* and *(iv)*, along with total order propagation of updates (property *(ii))* ensure that, if a transaction t commits after t' at some site, then t will commit after t' at all sites (SI PROPERTY 1 (Snapshot Read)). SI PROPERTY 2 (No Write-Write Conflicts) is enforced by property (i).

In order to be efficient, we avoid synchronisation among all replicas in the commitment phase, relaxing SI guarantees to enforce PSI.

The key difference is that in PSI when a replica executes a transaction it can commit it right away without having to wait for the write-set to be propagated. Moreover, in PSI, the write-set propagation is done in causal order and not with ABCAST, i.e., in total order. This modification provides the ability to commit non-conflicting transactions in a different order at different sites. The lost-update anomaly [12] is prevented by the causal order broadcast of write-sets. A lost update occurs when two transactions read the same object and then modify this object independently. The transaction that is committed last overwrites the changes made by the earlier transaction.

An event e *causally precedes* f, written $e \rightarrow f$, whenever the same server executes e before f or if there is an event g such that $e \rightarrow g \wedge g \rightarrow f$ (transitive closure). In Gargamel the causal relation is depicted in the transaction graph by the back dependency relation. If a transaction t'' is reachable in the graph from a transaction t' then there is a causal dependency between t' and t''. The fact that t' and t'' can be at more than one step of distance captures the transitive closure of the causal order. Causal order is a generalization of FIFO order and implies a partial order: two transactions not related in the graph (i.e., they have disjoint write-sets) are not ordered each other.

To ensure PSI we modify the Gargamel properties as follows:

(i) Conflicting transactions execute sequentially. This implies *a fortiori* that a transaction commits only if its updates do not conflict with a concurrent transaction.

(ii) Workers propagate their updates using causal order broadcast. When a replica receives the write-set of some transaction, it is guaranteed to have already received the write-sets of all preceding transactions.

(iii) Each replica's database engine ensures SI locally. Causal order propagation of updates ensures that a transaction t cannot commit after t' at any site if t' has observed the updates of t.

(iv) A worker does not start an update transaction until it has received the updates of preceding transactions.

(i) is unchanged. PSI PROPERTY 1 (Site Snapshot Read) is ensured by *(ii)* and *(iii)*. PSI PROPERTY 3 (Commit Causality Across Sites) is ensured by *(iv)*.

3.4 Conclusion

The Gargamel approach is designed to ensure scalability of a fully-replicated distributed DB. Our experimental study confirms that this is the case (see Chapter 7). These goals are obtained by a pessimistic a priori concurrency control, which eliminates aborting transactions. If the classifier does not have false negatives, it avoids the certification phase after the transaction execution. The performance and resource utilisation gain depend on classifier accuracy and on the workload characteristics. Gargamel offers the flexibility to implement multiple isolation levels to meet application requirements with a minimum synchronisation.

As Gargamel does the concurrency control in advance, before transaction execution, it needs to have an estimation of the transactions reads and writes, when the transaction is submitted to the system. Interactive transactions (i.e., clients send sequences of reads and writes commands encapsulated between a start- and a commit-transaction command) are not supported. We believe this loss of generality pays off, and it is not not over restrictive for OLTP applications.

Our proof-of-concept transaction classifier uses a simple static analysis of the workload. It checks over parameters of stored procedure call to predict conflicts. We exhibit a classifier for TPC-C that is sound, i.e., if a conflict exist it will be predicted, however, it can have false positives (it can predict conflicts that never appear at runtime). We have discussed the effect of an imperfect classifier: false positives imply that some transactions are serialised, even though they could execute in parallel. This results in lower performance than the theoretical maximum. We are considering extending our current approach with machine-learning techniques, learning from the classifier's past mistakes. However, this may also cause false negatives, which imposes certification af-

ter the execution in order to avoid to commit concurrent conflicting transactions.

Algorithm 1: Gargamel single-site scheduling protocol

```
1:  Variables:
2:      backDependencies, transactionQueue, graphHeads, targetWorker, precedingWorker
3:
4:  scheduleLocal(T_i)
5:      pre:   received ⟨SCHEDULE, T_i⟩ from client
6:      eff:   backDependencies = calculateBackDependencies(T_i)
7:             targetWorker = getExecutingWorker(T_i, backDependencies)
8:             send ⟨EXECUTE, T_i, backDependencies⟩ to targetWorker
9:
10: calculateBackDependencies(T_i)
11:     eff:   transactionQueue = graphHeads
12:            while !transactionQueue.isEmpty()
13:              T_j = transactionQueue.pool
14:              if conflict(T_i, T_j) then backDependencies.addT_i
15:              else transactionQueue.addAllT_i backDependencies
16:            return backDependencies
17:
18: getExecutingWorker(T_i, backDependencies)
19:            targetWorker = null
20:            if backDependencies.size() >= 1
21:              for allbackDependencies : precedingWorker
22:                if precedingWorker.closeTheChain then
23:                  targetWorker = precedingWorker.getExecutionWorker()
24:                  precedingWorker.closeTheChain = false
25:                  break
26:            if targetWorker == null then targetWorker = lessLoadedWorker
27:            targetWorker.closeTheChain() = true
28:            returntargetWorker
29:
```

Chapter 4

Multi-Site Gargamel

Contents

4.1 Introduction 46

4.2 System Architecture 47

4.3 Distributed Scheduling and Collision Resolution 47

4.3.1 Collisions 49

4.3.2 Collision and Synchronisation Protocol 53

4.3.3 Determinism and Duplicate Work 54

4.4 Fault Tolerance 56

4.4.1 Scheduler Failure 56

4.4.2 Node Failure 58

4.5 Conclusion 58

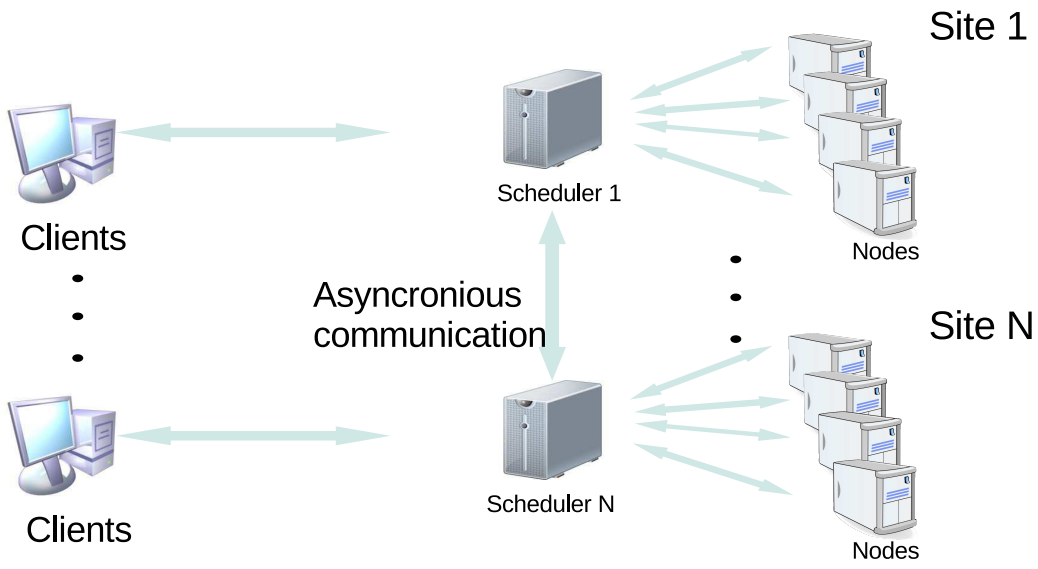


Figure 4.1: Multi-site system architecture

4.1 Introduction

In Chapter 3 we described a database architecture distributed over a single site (i.e., datacenter) focusing on the ability to scale and on optimal resource utilisation. These goals are met, thanks to a pessimistic concurrency control. This serialises conflicting transactions ahead of time to avoid aborts and spreads non-conflicting ones.

In this Chapter, we discuss a multi-site architecture, with a scheduler per site. This has the potential to lower client latency by connecting to a closer replicas. This requires schedulers to synchronise in order to avoid divergence. To avoid penalizing system throughput, the schedulers synchronise optimistically, off of the critical path.

4.2 System Architecture

In Gargamel multi-site, as illustrated in Figure 9.2, there are several sites, each with its own Gargamel scheduler and a local set of nodes.

A site might be, for instance, a powerful multicore within a datacenter, or a datacenter in a cloud. The important point is that the time to deliver a message sent from one site to another, the *inter-site message latency*, is much higher than the time from one of the schedulers to its local workers. We recall that a worker is one of the process accessing the database. In our experiments a node has one worker for each CPU.

Each scheduler manages transaction execution at its nodes as described in the previous Chapter: within a site, conflicting transactions are serialised and non-conflicting ones are spread among replicas.

As before, a scheduler receives transactions from local clients and sends them to its nodes for execution in parallel. Furthermore, a scheduler needs to synchronise its local view of the transaction graph with other schedulers to include conflicts with transaction schedulded at remotes schedulers. Synchronisation between schedulers is optimistic i.e., a scheduler first sends the transaction to a worker, then synchronises with other schedulers.

We envisage several classes where a multi-site configuration is useful. For instance, if the client-to-scheduler message latency is high, it may be beneficial to create a site close to the client. This helps to lower the client-perceived latency. Another case is when the workload exceeds the capacity of a single site; or when greater availability is required and replicas should be spread in multiple geographical locations.

4.3 Distributed Scheduling and Collision Resolution

Schedulers communicate asynchronously with one another, and are loosely synchronised. A scheduler can receive a transaction, either from a local client (we call this

a *home* transaction) or from another scheduler (a remote transaction). First consider a home transaction. The scheduler performs the normal scheduling algorithm described in Section 3.3. This is done without *a priori* synchronisation with the other schedulers. Once it has scheduled the remote transaction, the scheduler forwards it to all the other sites (asynchronously and reliably) along with its scheduling position. Schedulers use reliable FIFO broadcast to propagate transactions between them.

Once committed by the local database, the transaction's write-set is propagated to all other nodes (in both local and remote sites). The latter apply the write-set without re-executing the transaction [47]. Thus, although (as we shall see) a given transaction is scheduled at all sites, it nonetheless consumes computation resource at a single site only. This avoids duplicate work, and divides the load (Section 4.3.3).

Algorithm 2 describes the scheduling protocol in pseudocode. Logically, it can be divided into two parts: $scheduleLocal()$, called when a client sends a SCHEDULE message to its local scheduler, and $scheduleRemote()$ called when a scheduler sends a SCHEDULE_REMOTE message with its local dependencies to other schedulers. The $calculateBackDependencies()$ function (which computes whether the incoming transaction conflicts with a transaction in the local graph) and $getExecutingWorker()$ (which selects the worker that will execute the transaction) are the same as in Algorithm 1.

The variables of the scheduling protocol (omitted in the pseudocode for space reasons) are $localBackDependencies$ and $remoteBackDependencies$: the list of all the conflicting transactions in the local and remote graphs respectively. The $localBackDependencies$ list is sent along with the SCHEDULE_REMOTE message. The other variables: $transactionQueue$, $graphHeads$, $targetWorker$ and $precedingWorker$ are as in Algorithm 1. We recall that $transactionQueue$ is a queue used for breadth-first exploration of the graph; $graphHeads$ is a hash set containing all the "heads" of the graph, i.e., transactions without any front dependencies; $targetWorker$ is the ID of the worker chosen to execute the transaction; and $precedingWorker$ is a temporary variable, used to iterate over worker candidates for executing the incoming transaction.

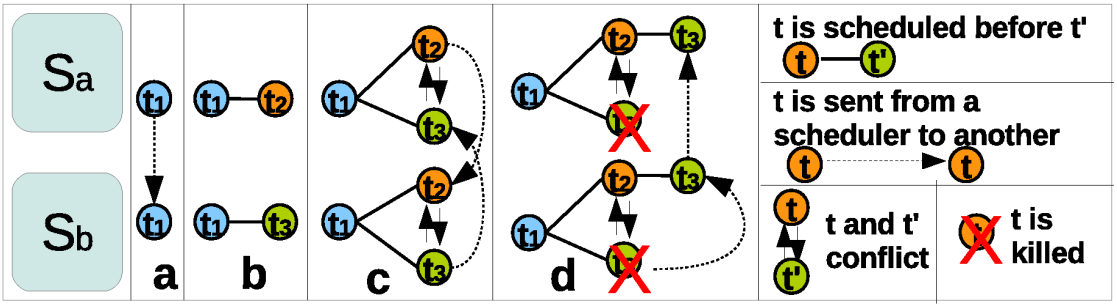
We analize now the algorithm's two main functions: $scheduleLocal()$ and $scheduleRemote()$. Note that $calculateBackDependencies(T_i)$ and $getExecutingWorker()$ are described in Section 3.3.

$scheduleLocal(T_i)$ When a client sends a transaction T_i to its local scheduler, the $scheduleLocal(T_i)$ function is called. The scheduler first computes T_i's back dependency (Line 3) and selects the executing worker (Line 4). If the worker is local (i.e., belongs to the same site) the transaction is sent to the worker (Lines 5–6). At the end the scheduler sends the transaction, along with its dependencies, to the other schedulers for agreement (Line 7).

$scheduleRemote(T_i, remoteBackDependencies)$ When a scheduler has scheduled a home transaction, it sends the transaction, along with its scheduling position, to all other schedulers (SCHEDULE_REMOTE message). Once SCHEDULE_REMOTE is received, the $scheduleRemote()$ function is called. The $scheduleRemote()$ function first computes the local back dependencies (Line 11), then checks for conflict with the descendants (front dependency) of the transactions in the local back dependency list (not in the remote one.) The descendants of a node n are nodes reachable from n through the front dependency lists. This check is done for all front dependency branches, until a conflict is found, or until the head of the graph is reached. Lines 13–16 performs a breadth-first exploration of potential conflicting descendants. Once all potential conflicts are found, we check if the execution worker is local to the scheduler that has received the SCHEDULE_REMOTE message (Lines 17–18). If yes, the transaction is sent for execution to the appropriate local worker.

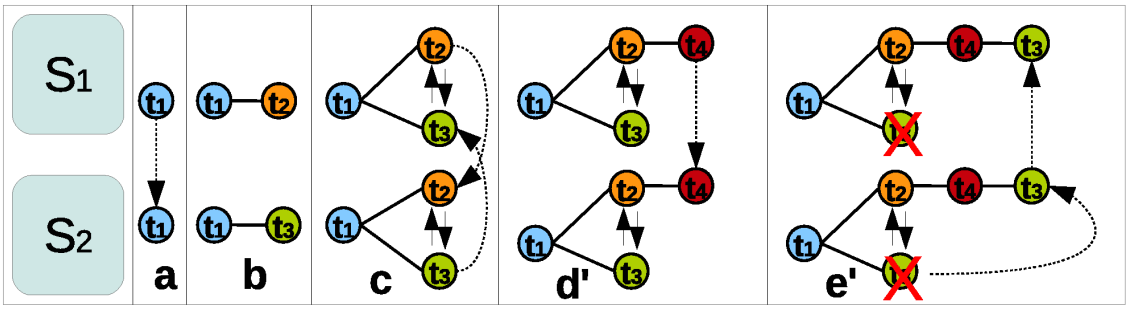
4.3.1 Collisions

When a scheduler S receives a remote transaction from a scheduler S', it schedules it according to the single-site scheduling algorithm of Chapter 3. When it compares its



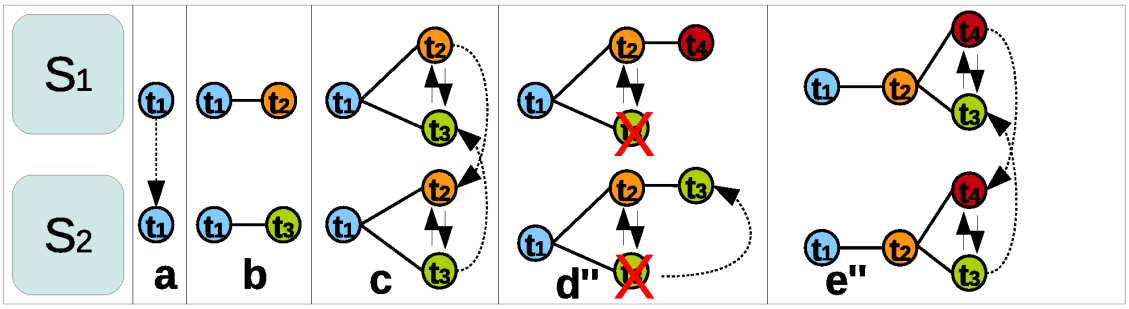
Two Sites S_1 and S_2, receive three conflicting transactions t_1, t_2, t_3. *(a)* S_1 receives t_1 and forwards it to S_2. *(b)* S_1 and S_2 receive t_2 and t_3 at the same time. *(c)* t_2 and t_3 are forwarded along with their scheduling position. S_1 and S_2 discover a collision. *(d)* S_1 and S_2 agree on the position of t_2 after t_1 and kill t_3, S_2 reschedules t_3 and forwards it to S_1.

Figure 4.2: Example collision.



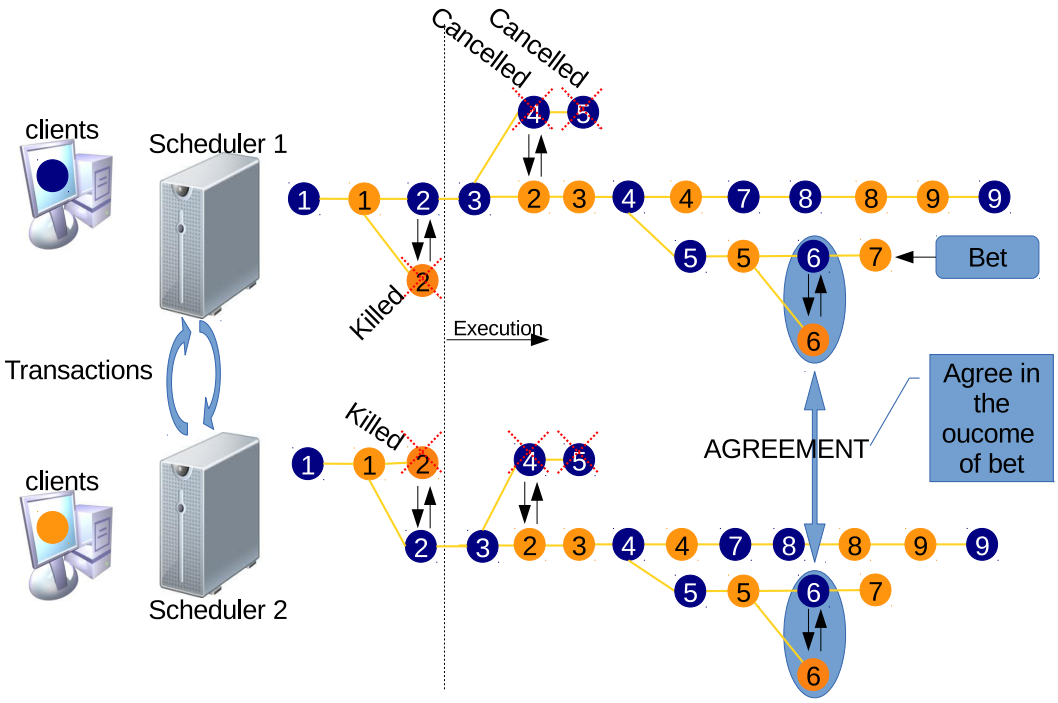
After the collision detection and before its resolution S_1 receives t_4. *(a)* (d') S_1 receives t_4, schedules it "betting" on t_2 and forwards it to S_2. *(b)* (e') S_1 and S_2 agree on the position of t_2 after t_1 and kill t_3, S_2 reschedules t_3 and forwards it to S_1.

Figure 4.3: Example bet.



While S_2 is rescheduling t_3 after t_2 S_1 schedules t_4 after t_2. A new conflict arises. *(a)* (d'') S_1 receives t_4 and S_2 reschedules t_3 at the same time. *(b)* (e'') t_3 and t_4 are forwarded along with their scheduling position. S_1 and S_2 discover a new collision.

Figure 4.4: Example collision on bet.



Example of transaction killed (agreement has been done after execution), cancelled (agreement before execution) and bets on forks not yet decided.

Figure 4.5: Cancellations, kills and bets at runtime

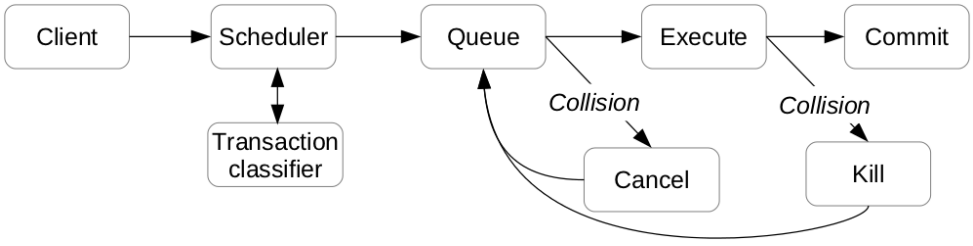


Figure 4.6: Transaction life-cycle

scheduling position with the position at S', it may find that the S' site has ordered the transaction differently. We call this situation a *collision*, S and S' must not both commit the transaction in different order (i.e., in a different position in the graph), otherwise they would diverge. Figures 4.2, 4.3 and 4.4 give some examples on how collisions happen and how they are managed.

Solving collisions requires a synchronisation protocol. The obvious solution would be to synchronise *a priori*, but this could be costly, since inter-site message latency is assumed high. For instance when sites are deployed across datacenters the WAN latency in much higher than the Local Area Network (LAN) latency and cross-datacenter communications are more prone to failure and partitions.

Instead, Gargamel's synchronisation is optimistic. It executes in the background, off of the critical path.

As illustrated in Figure 4.6, a collision may occur at different times in the life-cycle of a transaction: If the collision is detected after the transaction is queued, but before it is executed, it is simply re-queued in its correct position. We call this a *cancellation*. If the collision is received after the transaction starts, the offending transaction is forcefully aborted.[1] We call this *killing* the transaction. A transaction may not commit before the collision/non-collision information is received. If this is the case, then commitment must wait. We call this situation a *stall*.

A cancelled transaction costs nothing in terms of throughput or response time. A killed transaction (either during execution or after a stall) costs lost work, this has an impact on both throughput and response time. A stall followed by a commit do not cost lost work, but this impacts throughput and response time. Our experiments, in Chapter 7, show that if message latency is small compared to the transaction incoming rate, collisions most often result in cancellations. Even if the system suffers from a high collision rate, the lost work remains small.

[1] Note that this is the only case where a transaction aborts, since Gargamel has eliminated all concurrency conflicts.

4.3.2 Collision and Synchronisation Protocol

A collision splits a chain into incompatible branches. In the example in Figure 4.2*(c)*, the collision between transactions $t_2 - t_3$ splits the chain into a branch $t_1 - t_2$ and a branch $t_1 - t_3$. Those branches must not be both executed, because t_1, t_2 and t_3 mutually conflict, and should be serialised.

When this occurs, schedulers must reach agreement on which branch to *confirm* (Figures 4.2*(d)*). To this effect, Gargamel runs a consensus protocol between schedulers. In the presence of several branches, the protocol confirms the longest; or, if equal, the one with the first transaction with the smallest ID.

A transaction in another branch is either cancelled or killed, as explained earlier. Cancelling or killing a transaction T also cancels all local transactions that depend (directly or indirectly) on T. Thus, Gargamel does not need to run the confirmation protocol for each transaction it aborts, but only for a subset composed by one transaction for each branch.

Scheduling a transaction that conflicts with two or more colliding transactions requires *betting* on which one will win the confirmation protocol. Indeed, if a transaction is appended to conflicting branches it will be cancelled when one of the conflicting branch is cancelled. A bet consists in appending the transaction to one of the branches, hoping that the chosen branch will be confirmed. In order to maximize the probability of winning the bet, a scheduler applies the same heuristic as the collision protocol, i.e., it bets on the longest branch, or, if equal, one the one with the smallest transaction ID.

In the example in Figure 4.3, S_1 receives t_4, which conflicts with both t_2 and t_3. S_1 may bet either the chain $t_1 - t_2 - t_4$ or on $t_1 - t_3 - t_4$. Suppose the former (by smallest ID rule); if t_2 is confirmed, then t_4 will also be confirmed and the bet was a good one. In the worst case t_2 is cancelled and cause the cancellation of t_4 or S_1 bet on t_2 at the same time that S_2 reschedule t_3 causing a new conflict. The latter case is illustrated by Figure 4.4.

Figure 4.5 gives an overview of cancellations, kills, bets and agreements. The fig-

ure shows two schedulers, scheduler1 and scheduler2, scheduling concurrently blue and yellow transactions respectively. When they identify a conflict between already-executed transactions (transactions 2 blue and 2 yellow in the figure) they kill one of the conflicting branch (2 yellow in the figure). When they identify a conflict between transactions not yet executed (transactions 4 blue and 2 yellow in the Figure) they cancel one of the branches (the branch containing transactions 4 and 5 blue in the figure). When they have to schedule an incoming transaction that conflicts with two colliding transactions not yet cancelled by the agreement protocol, they bet on one of them (in the figure they bet on transaction 6 blue to append transaction 7 yellow.)

4.3.3 Determinism and Duplicate Work

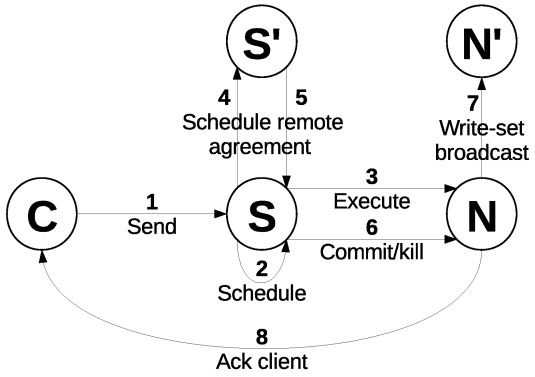
When a chain splits, as in Figure 3.3*(d)*, this results in a sequential execution of the common prefix (up to the split point) followed by a parallel execution (afterwards). If the common prefix was executed more than once non-deterministically, replicas might diverge.[2]

Gargamel takes this into account and avoids duplicate work by executing the prefix at a single node, and sending the resulting write-set to the other nodes. The other nodes simply apply the write-set to their local database, without re-executing the transaction code [47].

The node for which the first transaction in a branch is a home transaction is the only one that executes that branch.

Furthermore, to balance the load, multi-site Gargamel forwards incoming transactions from highly-loaded sites to less-loaded ones. When a site receives a home transaction transaction that starts a new branch, but no local worker is available, it forwards it to the least-loaded site instead of executing it locally. The sites estimate the remote load by counting the number of branches to be executed there.

[2] Non-determinism may occur either at the level of the transaction code, or at the level of the database. For instance, two different replicas might execute the same transaction against different DB snapshots.



Life-cycle of messages exchanged between clients, schedulers and nodes to execute a transaction. *(1)* The client sends a transaction to its home scheduler. *(2)* The scheduler schedules the transaction locally and *(3)* sends it for execution to a node. *(4)* The scheduler broadcasts the transaction, along with its back dependency to all other schedulers and *(5)* reaches an agreement on the position of the transaction in the dependencies graph. *(6)* The scheduler informs the node on the outcome of the agreement process (commit or kill). *(7)* In case of commit, the node broadcasts the write-set to other nodes and *(8)* acknowledges the client. In case of kill the node discards the execution.

Figure 4.7: Transaction message life-cycle

4.4 Fault Tolerance

In this section we discuss fault tolerance. We will just outline the Gragamel fault tolerance mechanism with very conservative assumptions. We consider crash faults [53], leading to a total loss in state. Machines do not crashes during the recovery process.

Multi-site Gargamel has a "natural" redundancy in the sense that the redundancy of schedulers and nodes is not for fault tolerance purpose but for other architectural considerations. This redundancy comes from the fact that each node has a full replica of the database, and each scheduler a full replica of the transactions graph. Nevertheless we can use this redundancy for crash recovery.

The system can recover as long as there is at least one correct scheduler with one correct node. For correctness, once a scheduler or a node suspects a machine to be crashed, it will discard all subsequent messages from this machine to avoid mistakenly suspected machines to come back in the system.

We address two different kind of failures: scheduler failure and node failure.

4.4.1 Scheduler Failure

In addition to being connected to a home scheduler, a client also maintains a list of alternate schedulers. When a client suspects that a scheduler S has failed, it and notifies a correct scheduler S' and sends S' the list ($transactionList$) of transactions it has sent to S and where not achieved. Recall that schedulers use reliable FIFO broadcast to propagate transactions between them, so if a scheduler has a remote (i.e., not coming from one of its clients) transaction in its local view, then eventually all correct schedulers will receive that same transaction.

A transaction t in $transactionList$ can be in one of three possible states:

i) t **is in the local view of** S'**:** this means that the scheduler has crashed after step 4 of Figure 4.7 (transaction propagation between schedulers). This implies that the

transaction has been already delivered for execution to some node (see Algorithm 2 and Figure 4.7). In this case, S' will take no actions. The client will eventually receive the reply for that transaction.

ii) t **is not in the local view of** S' **and is not scheduled for execution at any of the nodes of** S**:** this means that the scheduler has crashed before step 3 of Figure 4.7. The transaction is "lost" because, except for the client, none of the surviving nodes on the system knows about it. In this case, S' reschedule the transaction as a home transaction transaction.

iii) t **is not in the local view of** S' **and it is scheduled for execution at one of the nodes of** S**:** this means that S crashed after step 3 and before step 4 of Figure 4.7. The transaction has been scheduled and sent to a node for execution, but the SCHEDULE_REMOTE message was not sent to the other schedulers. In this case, S' retrieves the transaction and its back dependency (as calculated by S) and reschedules it locally in the same position (i.e., keeping the back dependency calculated by S), and sends SCHEDULE_REMOTE to other schedulers (including itself). This recovers the execution from the point at which was interrupted.

This procedure can be repeated until there is at least one correct scheduler in the system. To handle false positives in crash detection, if a scheduler takes a decision based on the fact that it has not received a transaction (as seen in the above paragraph the scheduler decision depends on weather or not a remote transaction is in its local view) and then it receives the transaction, it will simply discard the transaction and reschedule it (point ii of the above paragraph). Similarly, if a scheduler recovers after being detected as failed (or it was not failed at all), it will rejoin the system as a new machine, i.e., with an empty transaction graph.

Algorithm 3 shows the pseudocode for the scheduler recovery protocol. The protocol is initiated by a client that suspect the crash of a scheduler acknowledging another scheduler (Lines 6–7). In this step the client uses two variables: $schedulers$, containing

the list of all the schedulers in the system and $pendingTransactions$, a list containing all transactions submitted by the client for execution and for which it has not yet received a reply (i.e., the commit message). In Line 6 the client picks up a scheduler other than the failed one, and in Line 7 it sends the new scheduler a message containing all pending transactions and the ID of the failed scheduler.

When a correct scheduler receives the information that a scheduler has failed (Lines 9–12) it sends a recovery message to all nodes of the failed scheduler, containing the list of pending transactions not locally scheduled at the correct scheduler.

4.4.2 Node Failure

When a Scheduler suspects a node failure, it fetches from the graph the list of transactions sent to that node for execution and checks on the survivor nodes which write-sets have not been received. It then reschedules for execution transactions that have not been received by survivor nodes. Notice that nodes send reply to clients after broadcasting the write-set to other nodes, otherwise in case of failure clients can receive more than one reply to the same transaction. Algorithm 4 shows the pseudocode of the recovery protocol when a node fails. As in Algorithm 3, the protocol is initiated by a client that reacts to a suspicious of a node informing the failed node's home scheduler (Line 7).

When the scheduler receives the RECOVER message (Line 10) it checks for all "lost" pending transactions (i.e., transactions sent for execution but not known to any correct node) and reschedules them for execution in a correct node (Lines 11–12).

4.5 Conclusion

Multi-site Gargamel allows several geo-replicated sites, each composed by a scheduler and a set of nodes, to proceed in parallel. Each site receives transactions from local clients and executes them at local nodes. Synchronisation among sites on the execution

order is done optimistically, off of the critical path.

Multi-site Gargamel is suitable to lower client perceived latency by putting schedulers closer to them, to improve availability spreading schedulers in multiple geographical locations and to expand the system when the workload exceed the capacity of a single site.

We have described the system architecture, the distributed scheduling and collision resolution algorithm and outlined the fault tolerance.

We evaluated its performances and benefits by a discrete event simulator. Simulation results are presented in Chapter 5.

Algorithm 2: Gargamel multi-site scheduling protocol

```
1:  scheduleLocal(T_i)
2:      pre:  received ⟨SCHEDULE, T_i⟩ from client
3:      eff:  localBackDependencies = calculateBackDependencies(T_i)
4:            targetWorker = getExecutingWorker(T_i, localBackDependencies)
5:            if targetWorker is local then
6:               send ⟨EXECUTE, T_i, localBackDependencies⟩ to targetWorker
7:            send ⟨SCHEDULE_REMOTE, T_i, localBackDependencies⟩ to schedulers
8:
9:  scheduleRemote(T_i, remoteBackDependencies)
10:     pre:  received ⟨SCHEDULE_REMOTE, T_i, remoteBackDependencies⟩ from scheduler
11:     eff:  localBackDependencies = calculateBackDependencies(T_i)
12:           transactionQueue = (localBackDependencies − remoteBackDependencies) front
      dependencies
13:           while !transactionQueue is empty
14:              T_j = transactionQueue.pool
15:              if conflict(T_i, T_j) then add conflict T_i, T_j
16:              else transactionQueue.addAll T_i front dependencies
17:           if T_i worker is local then
18:              send ⟨EXECUTE, T_i, remoteBackDependencies⟩ to worker
19:
20: calculateBackDependencies(T_i)
21:     eff:  transactionQueue = graphHeads
22:           while !transactionQueue is empty
23:              T_j = transactionQueue.pool
24:              if conflict(T_i, T_j) then backDependencies.add T_i
25:              else transactionQueue.addAll T_i back dependencies
26:           return backDependencies
27:
28: getExecutingWorker(T_i, backDependencies)
29:           targetWorker = null
30:           if backDependencies.size() >= 1
31:             for allbackDependencies : precedingWorker
32:               if precedingWorker.closeTheChain then
33:                 targetWorker = precedingWorker.getExecutionWorker()
34:                 precedingWorker.closeTheChain = false
35:                 break
36:           if targetWorker == null then targetWorker = lessLoadedWorker
37:           targetWorker.closeTheChain() = true
38:           returntargetWorker
39:
```

Algorithm 3: Gargamel scheduler recovery protocol

Variables:

transactions, *pending*, *tuple*<$t, tBackDependencies$>, *pendingTransactions*, *schedulers*, *scheduledTransactions*

startRecovery()

pre: Scheduler S is suspected

eff: select $S' \mid S' \in schedulers \wedge S'$ is correct $\wedge\ S' \neq S$

send $\langle$RECOVERSCHEDULER, *pendingTransactions*, $S\rangle$ *to* S'

recoverScheduler(S, *pendingTransactions*)

pre: *received* $\langle$RECOVERSCHEDULER, S, *pendingTransactions*$\rangle$ *from client*

eff: $\forall$ Node $N \in S$

send $\langle$RECOVERNODE, *pendingTransactions* $-$ *scheduledTransactions*$\rangle$ *to* N

recoverNode(*transactions*)

pre: *received* $\langle$RECOVERNODE, S, *transactions*$\rangle$ *from Scheduler*

eff: $\forall t \in$ *transactions add*(t, $t.backDependencies$) *to tuple*

send $\langle$TRANSACTIONSRECOVERED, *tuple*$\rangle$ *to* S

transactionsRecovered(*tuple*)

pre: *received* $\langle$TRANSACTIONSRECOVERED, *tuple*$\rangle$ *from node*

eff: $\forall < t, tBackDependencies > \in$ *tuple*

$addToGraph(< t, tBackDependencies >)$

send $\langle$SCHEDULE_REMOTE, $< t, tBackDependencies >\rangle$ *to schedulers*

$pendingTransactions - \{t\}$

if received all reply **then**

$\forall pending \mid pending \in pendingTransactions$

$pendingBackDependencies = scheduleLocal(pending)$

$scheduleRemote(pending, pendingBackDependencies)$

Algorithm 4: Gargamel node recovery protocol

```
1:  Variables:
2:      pendingTransactions
3:
4:  startRecovery()
5:      pre:  Node N is suspected
6:      eff:  select S | S is the home scheduler of N
7:            send ⟨RECOVER, pendingTransactions, N⟩ to S
8:
9:  recover(N, pendingTransactions)
10:     pre:  received ⟨RECOVER, N, pendingTransactions⟩ from client
11:     eff:  ∀t | t ∈ pendingTransactions ∧ t writeSet is not propagated
12:           reschedule t on a correct node
13:
```

Chapter 5

Simulation

Contents

5.1 Simulation Model 64

5.1.1 TPC-C 64

5.1.2 TPC-E 67

5.1.3 Round-Robin, Centralised-Writes and Tashkent+ Simulation . . . 68

5.2 Simulation Results 69

5.2.1 Single-Site Performance 72

5.2.2 Single-Site Resource Utilisation, Bounded Workers 73

5.2.3 Single-Site Resource Utilisation, Unbounded Workers 76

5.2.4 Multi-Site Gargamel 78

5.2.5 Adaptation to The Number Of Workers 86

5.3 Conclusion 88

We implement a discrete event simulator to evaluate the advantages and limits of the Gargamel design and to understand the limitations and requirements before proceed with a full implementation (in later chapter).

The aim of the simulation is to evaluate (i) response time and throughput of update transactions, (ii) overhead, (iii) resources requirements, (iv) the impact of collisions in the multi-site case, (v) the accuracy of the classifier.

5.1 Simulation Model

In this section we present the simulation model and describe the benchmarks we use. The experiments and results are discussed in Section 5.2.

We consider two different workloads, TPC-C [66] and TPC-E [67]. They have different characteristics; in particular, transactions of the former are longer and more write-intensive than the latter.

At the beginning of a simulation run, clients submit a stream of transactions, and the scheduler sends them to nodes. When the incoming rate exceed capacity, a backlog of queued transactions grows. At some point (28 seconds in the single site simulation, 3,5 seconds in the multi-site simulation) clients stop submitting, and the system empty the backlog. The experiment ends when the last transaction has terminated.

Unless specified otherwise, all simulations use the parameters presented in Table 5.1.

The transaction classifier is based on a static analysis of transaction text. The TPC-C transaction parameters give sufficient information to avoid false positives almost entirely. In contrast, our TPC-E classifier exhibits a substantial amount of false positives.

5.1.1 TPC-C

TPC-C benchmark is composed of five type of transactions: New Order (NO), Payment (P), Order Status (OS), Delivery (D), and Stock Level (SL). 92% of the workload consists of update read-write transactions; the remaining 8% are read-only (OS and SL). Under Snapshot Isolation, read-only transactions do not conflict.

Parameter	**Value**		
Default number of workers	100		
Default incoming rate	150 trans/s		
Default load (single-site)	150 trans/s for ~ 28 s		
Default load (multi-site)	150 trans/s for ~ 3.5 s		
Warehouses (TPC-C)	10		
	(ms)	**Mean**	**Variance**
Inter-site msg latency	60	1	.005
Site-worker msg latency	.06	1	.005
Consensus latency	180	1	.01
Client-site msg latency	0		
Apply write-set	0		
TPC-C	**Duration**	**Mean**	**Variance**
transactions [7, 33, 66]	(ms)		
New Order	700	1	.025
Payment	660	1	.028
Order Status	680	1	.028
Delivery	660	1	.035
Stack Level	1010	1	.022
TPC-E	**Duration**	**Mean**	**Variance**
transactions [34, 67]	(ms)		
Broker Volume	30	1	.025
Customer Position	20	1	.025
Market Feed	20	1	.025
Market Watch	30	1	.025
Security Detail	10	1	.025
Trade Lookup	110	1	.025
Trade Order	50	1	.025
Trade Result	60	1	.025
Trade Status	10	1	.025
Trade Update	130	1	.025

Transaction durations are generated using a Gaussian distribution with the indicated mean and variance. The numerical parameters of TPC-C and TPC-E are taken from the referenced measurements [33, 34].

Table 5.1: Simulation parameters

Transaction pairs	Conflict condition
$NO(w_1, d_1, I_1) \times NO(w_2, d_2, I_2)$	$(w_1 = w_2 \wedge d_1 = d_2) \vee I_1 \cap I_2 \neq \emptyset$
$P(w_1, c_1, cw_1, cd_1) \times P(w_2, c_2, cw_2, cd_2)$	$(w_1 = w_2) \vee ((cw_1 = cw_2 \wedge cd_1 = cd_2) \wedge (c_1 = c_2))$
$D(w_1) \times D(w_2)$	$w_1 = w_2$
$D(w_1) \times P(w_2, c_2, cw_2, cd_2)$	$w_1 = cw_2$

The subscripts represent two concurrent transactions. Please refer to Section 5.1.1 for an explanation of variable names.

Table 5.2: Conflicts of TPC-C under SI

A $NO(w, d, I)$ transaction adds a complete order to a warehouse. Its parameters are a warehouse w, a district d, and a list of items I. Each item $i(w', d') \in I$ has two parameters: its warehouse ID w' and the item ID d'. An I list contains between 5 and 15 elements. NO transactions occur with high frequency and relatively costly; they dominate the workload.

The parameters of a Payment transaction $P(w, c, cw, cd)$ are a warehouse ID w, a customer c, a customer warehouse cw, and a customer district cd. The customer c is selected 60% of the time by name, and 40% of time by unique identifier. Homonyms are possible in the former case.

The single parameter of a Delivery transaction D_w is warehouse ID w.

In our classifier, two transactions are considered to conflict: (i) if they update the same column of a table, (ii) unless it is clear from the analysis that they never update the same row. In the case of TPC-C, conflicts may happen between pairs of the same transaction type (NO and NO, P and P, D and D) and between P and D transactions. Table 6.1 shows which transactions conflict according to their parameters.

Since the customer of a Payment transaction is selected 60% of the time by name and homonyms cannot be checked statically, transaction classification admits false positives between two Payment transactions and between a Payment and a Delivery transaction. If the customer is identified by name, the classifier conservatively assumes that a conflict

is possible, false positives are possible. When a false positive arises, the resulting schedule is not optimal, because it serialises two (Payment) transactions that in fact would not need to be serialised.

Our classifier does not suffer from false positives induced by homonyms in Delivery transaction pairs, because in Delivery transaction the customer selection is based on the lowest ID (representing the oldest NO) which are unique for a given snapshot.

5.1.2 TPC-E

Our second benchmark, TPC-E [67], is another standard benchmark for OLTP transactions. TPC-E is interesting because the workload has different access patterns and transaction execution time than TPC-C. The corresponding parameters in Table 5.1 are taken from actual TPC-E results [34].

It is composed of twelve transaction types. Six are read-only and six are read/write. Type *Trade-Cleanup* executes only once, and has little influence on overall performance. Type *Data-Maintenance* simulates administrative updates, and is not significant. Excluding these non-significant transaction types, the TPC-E workload consists of only 23.1% of update transactions vs. 76.9% read-only.

As above, our TPC-E transaction classifier is based on a static analysis. Under SI, only update transactions can conflict. Furthermore, Trade Order transactions never conflict, because they only add new unique information (new rows) and never update existing information. Therefore, this discussion focuses on the Market Feed (MF), Trade Result (TR) and Trade Update (TU) transaction types.

The transaction parameters of MF and TU are sufficient to accurately check their conflicts. However, the TR parameter *tradeID* does not permit to statically check wich fields will be affected (because the *customerID* and the *brokerID* are calculated at runtime), thus conflict prediction for TR is not accurate.

In Section 5.2 we will see how this inaccuracy impacts performance.

5.1.3 Round-Robin, Centralised-Writes and Tashkent+ Simulation

Our simulations compare Gargamel with a simple Round-Robin scheduler, and with the state-of-the-art Tashkent+ [24]. Tashkent+ is a memory-aware load balancing algorithm that exploits knowledge of the working sets of transactions to execute them in memory, thereby reducing disk I/O. We have not compared Tashkent+ with the implementation of Gargamel because in our prototype we have used an in-memory database, therefore there is no point in optimising disk eccesses. We have also run some experiments with the database on the disk to check the memory usage and we have measured that nodes resident in Amazon Elastic Compute Cloud (EC2) "m3.medium" instances never use all their memory.

When possible, we also compare Gargamel with a centralised-writes approach.

Round-Robin aims to maximise throughput by running as many transactions as possible in parallel. It works as follows. Each incoming transaction is assigned to a worker in equal portions and in circular order. Because concurrent transactions may conflict, Round-Robin suffers from a lot of abort-restarts, i.e., wasted work.

A Centralized-Write system runs read-only transactions concurrently, but serialises all update transactions at a single worker, in order to avoid wasted work. It can be considered as an idealized version of Ganymed [48]. Centralized-Write is simulated in Gargamel by classifying all update transactions as mutually-conflicting. Therefore, Gargamel puts all update transactions into a same queue, executed by a same worker. Our simulations shows that Centralized-Writes is overly conservative on standard benchmarks.

Like Round-Robin, Tashkent+ aims to maximise throughput, but optimises the assignment of transactions to workers, by ensuring that the working-set of each group of transactions sent to a worker fits into the worker's main memory. To balance the load, Tashkent+ monitors each group's CPU and disk usage, and rearranges groups, by moving workers from the least loaded group to the most loaded group.

Tashkent+ estimates the working set of an incoming transaction by examining the database's execution plan. Our simulated Tashkent+ extracts the execution plan from TPCC-UVA [41], an open source TPC-C implementation.

Our simulator implements the Tashkent+ group allocation/re-allocation algorithm as described in the literature [24]. Since CPU and disk usage are not significant in this simulation, we estimate load by the ratio of busy to available workers. Replica allocation and re-allocation are implemented in such a way that balance remains optimal all the time. Our simulations are favorable to Tashkent+ because we assume that re-allocation has no cost.

As the literature shows that Tashkent+ improves performance by reducing disk access, our simulation takes this into account by reducing the duration of every transaction by 10% under Tashkent+. However, our simulations hereafter show that Tashkent+ suffers from aborts (lost work) under TPC-C.

5.2 Simulation Results

The simulations cover the following areas:

Single-site performance: We measure transaction throughput, response time, and amount of resources consumed, comparing single-site Gargamel to Tashkent+ and Round-Robin. When relevant, we compare also with Centralized-Write. This set of experiments is based on TPC-C, varying transaction incoming rate.

Multi-site system behaviour: In the multi-site case, our focus is to understand whether collisions are an issue. Therefore, we compare two workloads with different collision behaviours, TPC-C and TPC-E, varying the number of sites.

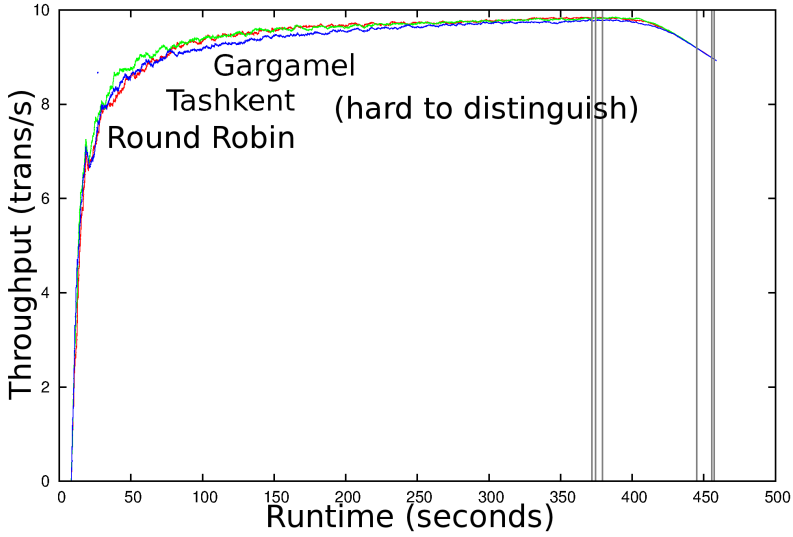


Figure 5.1: Throughput (TPC-C, 10 trans/s).

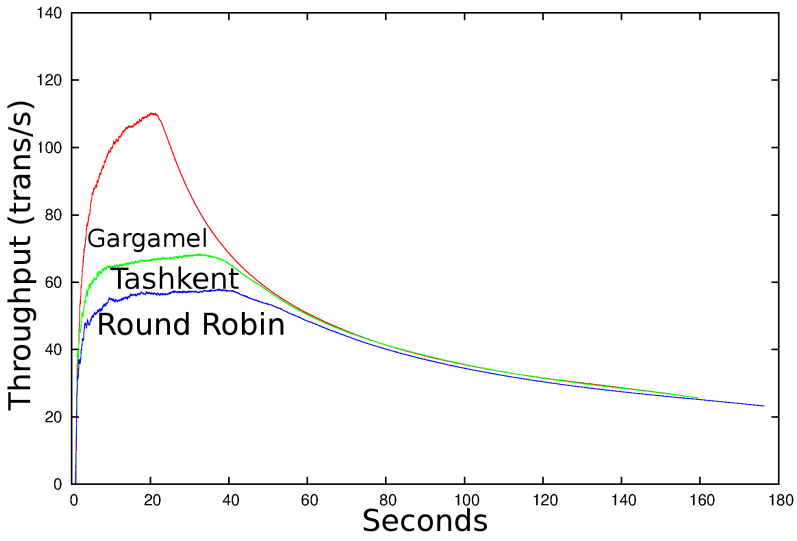


Figure 5.2: Throughput (TPC-C, 200 trans/s).

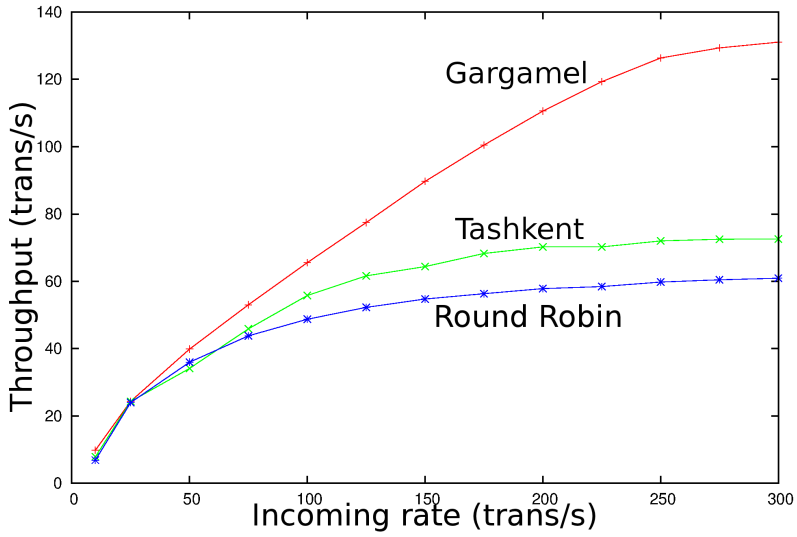


Figure 5.3: Maximal throughput (TPC-C).

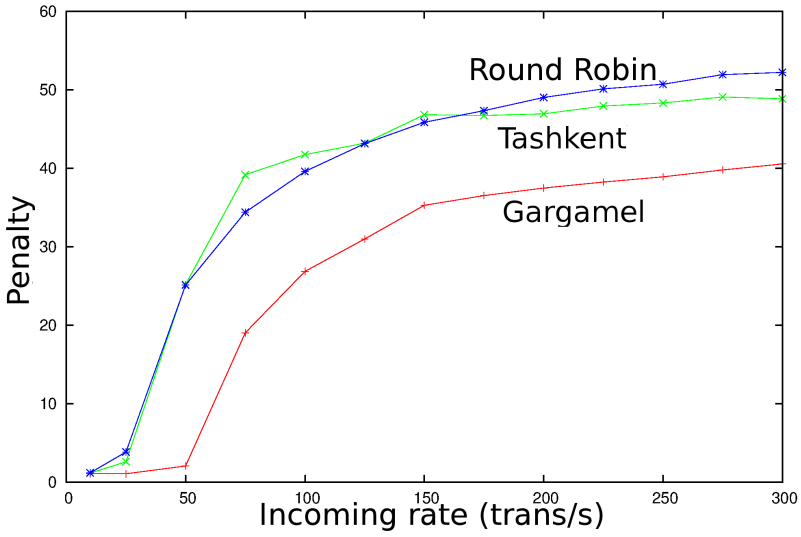


Figure 5.4: Penalty ratio (TPC-C).

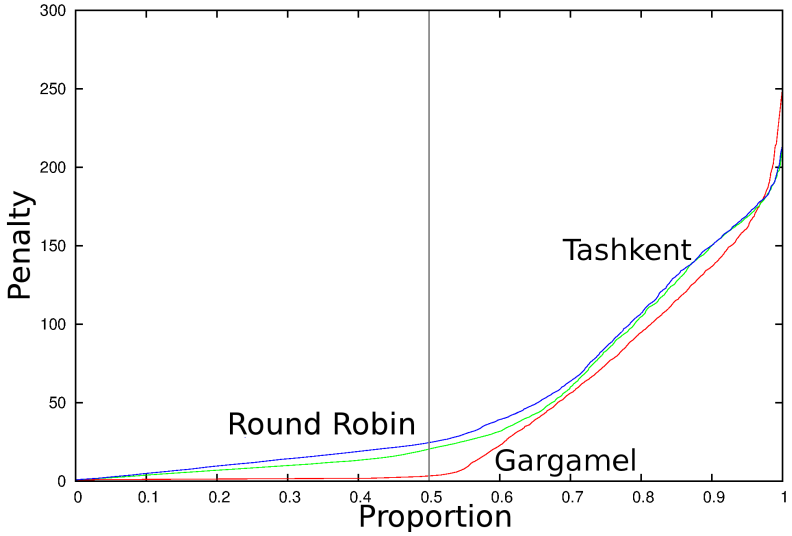


Figure 5.5: Cumulative distribution function of penalty ratio (TPC-C).

5.2.1 Single-Site Performance

When the incoming rate is low, parallelism is low, therefore conflicts are not an issue. In such a situation, all schedulers are basically equivalent. For instance, Gargamel will schedule each incoming transaction to a new chain and executes it immediately. Results presented in Figure 5.1 confirm that at 10 trans/s (transactions per second) Gargamel, Tashkent+ and Round-Robin have similar throughput. A vertical line represents the point at which transactions stop arriving.

Things get more interesting at high incoming rates. Figure 5.2 compares the throughput of the three systems at 200 trans/s. Figure 5.3 shows maximum throughput, varying the incoming rate. The two figures show that Gargamel exhibits a significant improvement over the other systems during the first phase (while clients submit transactions). In the second phase (when no more transactions are submitted), parallelism decreases and the throughput of all three systems decreases consequently.

Figure 5.3 shows that the improvement of Gargamel over Tashkent+ and Round-

Robin grows with the incoming rate. At 300 trans/s, Gargamel saturates the available workers, and, at constant number of workers, an increase in incoming rate does not provide any improvement.

We estimate response time by measuring the "penalty ratio," i.e., response time (time elapsed between transaction submission and transaction commitment) over transaction duration. The lower the penalty ratio, the lower the scheduling delays suffered by clients. Figures 5.4 and 5.5 show the penalty ratio and its CDF, comparing Gargamel, Tashkent+ and Round-Robin. Gargamel's penalty is approximately 20% lower than Tashkent+ and Round-Robin. 51% of the transactions in Gargamel suffer a penalty of 4 or less, whereas this is the case of only 10% of transactions (approximately) in the competing systems.

The speedup is estimated by dividing the sequential execution time by the total execution time. The speed-up improvement is low (Figure 5.6), because although most transactions execute with little delay (as shown by the penalty ratio CDF), the longest-waiting transaction is delayed almost identically in all three systems. This is due to the fact that conflicting transactions must be serialised anyway and a long chain of mutually conflicting transactions appears.

5.2.2 Single-Site Resource Utilisation, Bounded Workers

Our next experiments examine resource utilisation and queue size in our systems.

The bottom part of Figure 5.7 shows the number of busy workers as the simulation advances in time. The top part shows the number of queued transactions (note that for readability the scale of the y axis differs between the bottom and the top).

For the default number of workers (100), at the default incoming rate (150 trans/s), Gargamel always finds an available worker, and queue size is close to zero. This means that at all times. the number of parallelisable transactions remains under the number of workers.

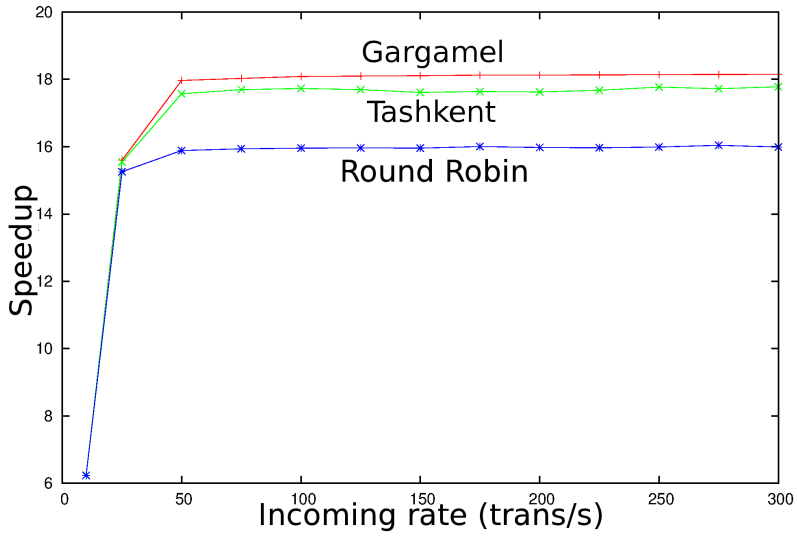


Figure 5.6: Speedup for TPC-C

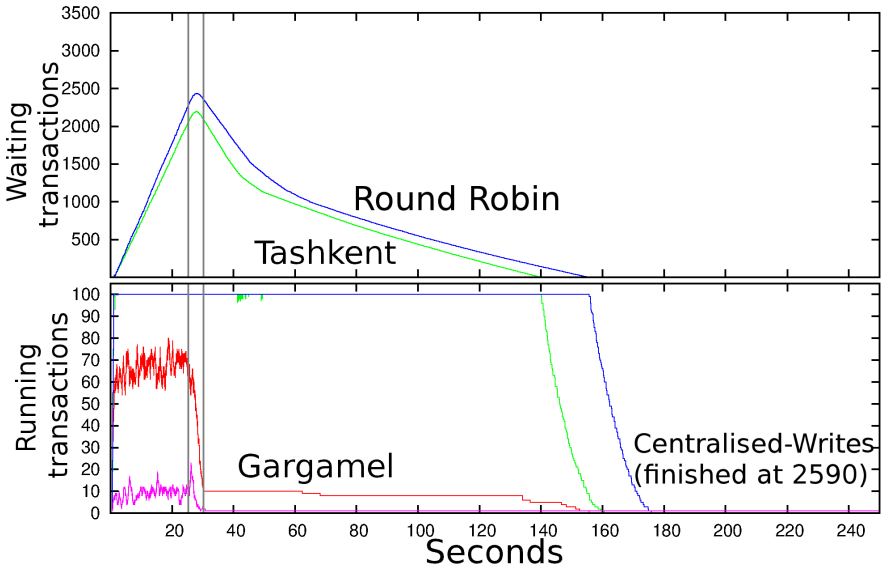


Figure 5.7: Resource utilisation for TPC-C at 150 trans/s

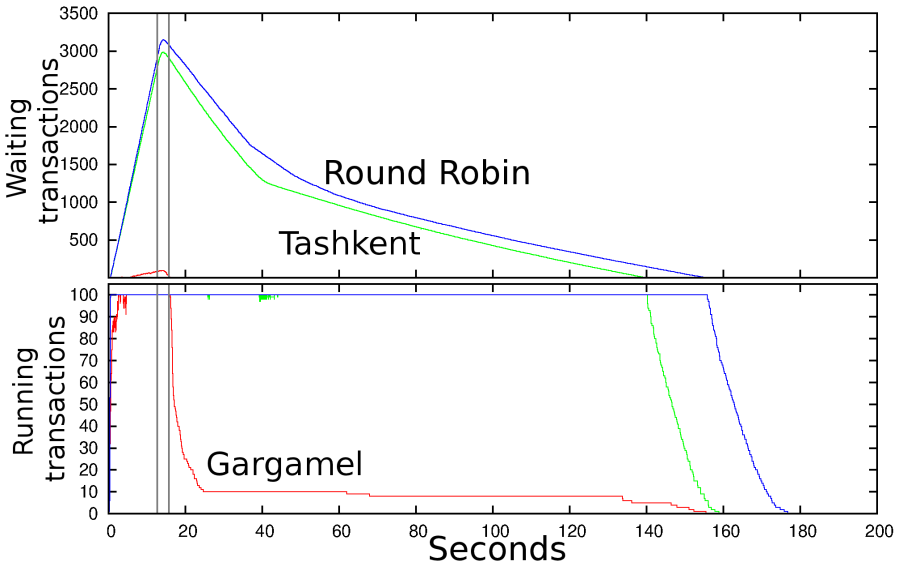


Figure 5.8: Resource utilisation for TPC-C at 300 trans/s

In contrast, the policy of both Tashkent+ and Round-Robin is to execute as many transactions as possible in parallel, as soon as they arrive. However, since many of those transactions conflict, there are many aborts, and they do not make progress. They quickly saturate the number of workers; incoming transactions are delayed, and queue size grows rapidly.

In the second phase, after transactions stop arriving (the incoming rate goes to zero, represented by vertical lines), Gargamel frees most of the workers. Indeed, at this point, all the read-only and non-conflicting transactions have finished executing; Gargamel only needs a few workers for the remaining long chains of conflicting transactions.

In contrast, Tashkent+ and Round-Robin continue to saturate the workers by attempting to parallelise conflicting transactions. At some point during the second phase, Tashkent+ re-assigns groups and continues to empty the queue of waiting transactions more slowly than before the group reassignment. This is because, at this point, all the read-only and non-conflicting transactions have terminated.

We have also simulated a rate of 300 trans/s (Figure 5.8). Even for Gargamel the load is too high for the default number of workers, and Gargamel builds a (very small) queue during the first phase.

5.2.3 Single-Site Resource Utilisation, Unbounded Workers

We now consider a system where the number of workers is unbounded, e.g., an elastic cloud computing environment. In this case, both Tashkent+ and Round-Robin mobilise a much higher amount of resources than Gargamel. Figure 5.9 shows that, at the end of the first phase, Tashkent+ needs 1500 concurrent workers, whereas Gargamel needs fewer than 100.

This translate directly into monetary cost. Considering that EC2 advertises a cost of approximately 0,1 euro instance/hour [35], Figure 5.10 plots the cost of the three systems, varying the incoming rate, with both the default number of workers, and with

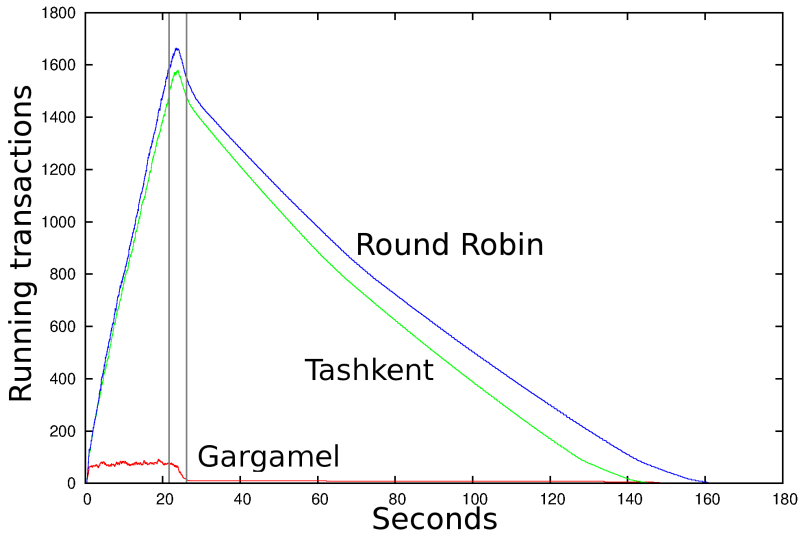


Figure 5.9: Resource utilisation, unbounded workers (TPC-C, 175 trans/s).

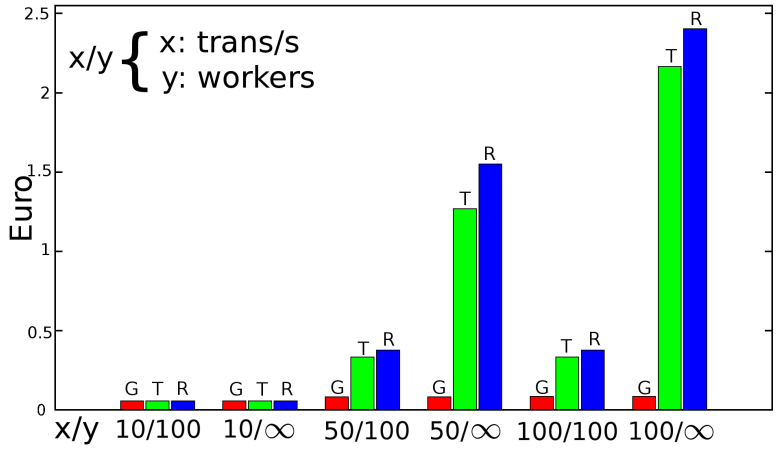


Figure 5.10: Cost comparison

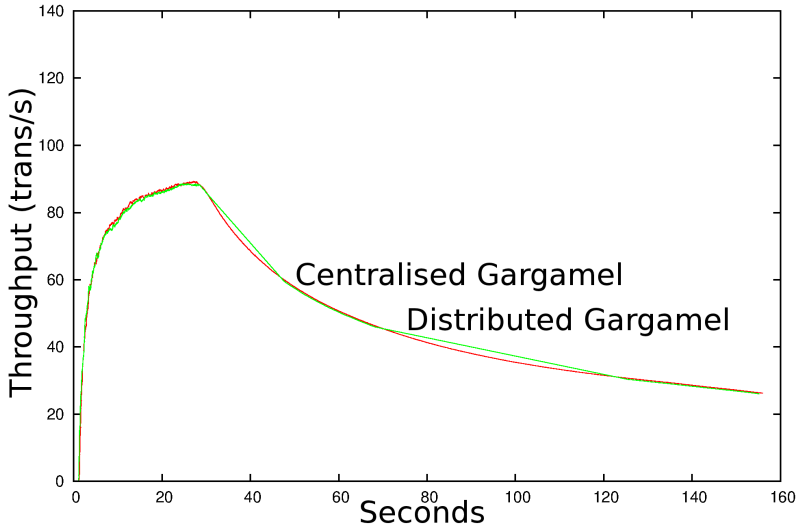


Figure 5.11: Throughput (TPC-C).

unbounded resources. At low incoming rate, all systems use the same small amount of resources. As the rate increases, Tashkent+ and Round-Robin use as many workers as possible in order to maximise parallelism. With bounded workers, once all workers are in use, the cost of Tashkent+ and Round-Robin remains the same, even if the incoming rate increases; if the number of workers is unbounded, the resource usage of Tashkent+/Round-Robin is proportional to the incoming rate. At 100 trans/s, with unbounded workers, Gargamel is 25 times cheaper than Tashkent+.

5.2.4 Multi-Site Gargamel

We compare the performance of single-site vs. multi-site Gargamel, to evaluate if the latter is advantageous. In order to ensure the comparison is fair, the makeup of the two systems is similar. We assume a set-up with two data centres, with 50 workers each. One-way message latency within a data centre is 0.06 ms, a typical value for a LAN, and it is 60 ms between data centres, which is a realistic value for a transatlantic WAN

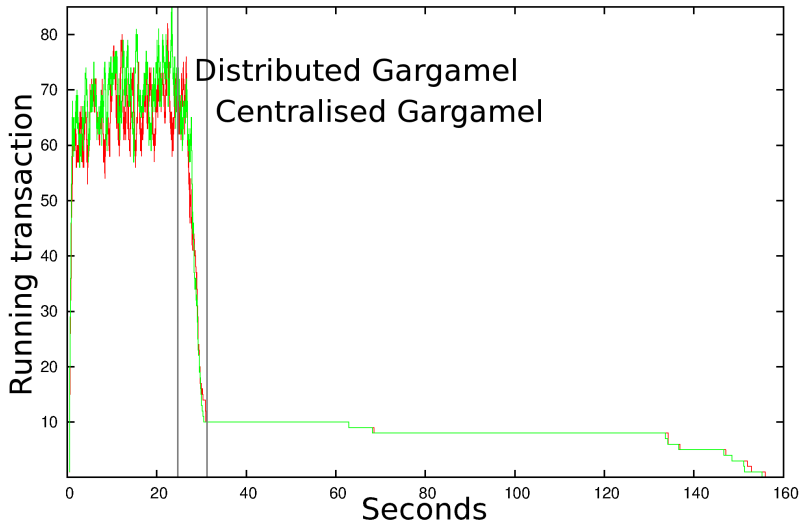


Figure 5.12: Resource utilisation (TPC-C).

link. In the single-site configuration, there is a single scheduler in one of the data centres, whereas each data centre has its own scheduler in the multi-site case.

Figures 5.11 and 5.12 compare the throughput and resource utilisation of single-site and multi-site configurations. In both cases, the two curves are barely distinguishable, showing that the overhead of multi-site is negligible.

5.2.4.1 Impact Of Collisions

As explained in Section 4.3, the optimistic approach to scheduling in the multi-site setting results in scheduling collisions. Collisions can be resolved either as an early *cancellation* i.e., reordering the transaction before starts its execution, at no cost (no lost work), or as a later *kill*, whereby the already-executing transaction is forcefully aborted and restarted. This section aims to measure the impact of collisions. In these experiments, there is a variable number of sites, each with an unbounded number of workers. We also vary the message latency between sites, thus increasing the probability of collision and

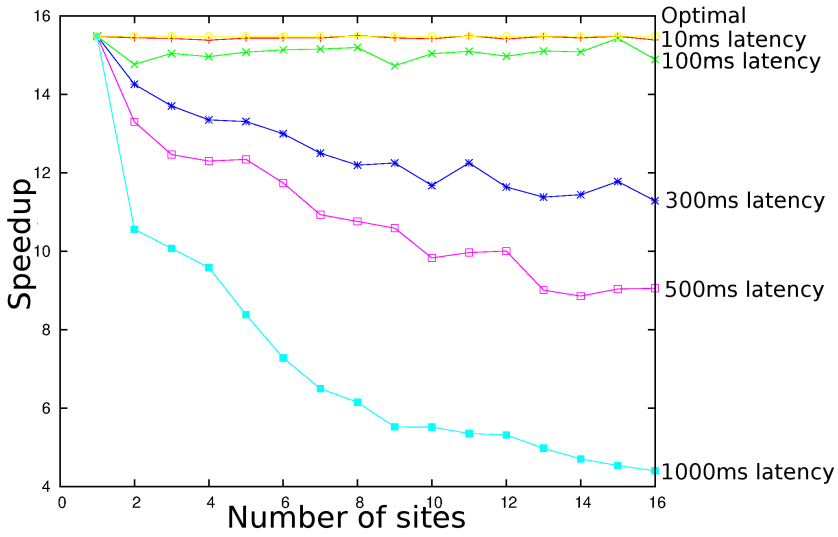


Figure 5.13: Impact of collisions for TPC-C

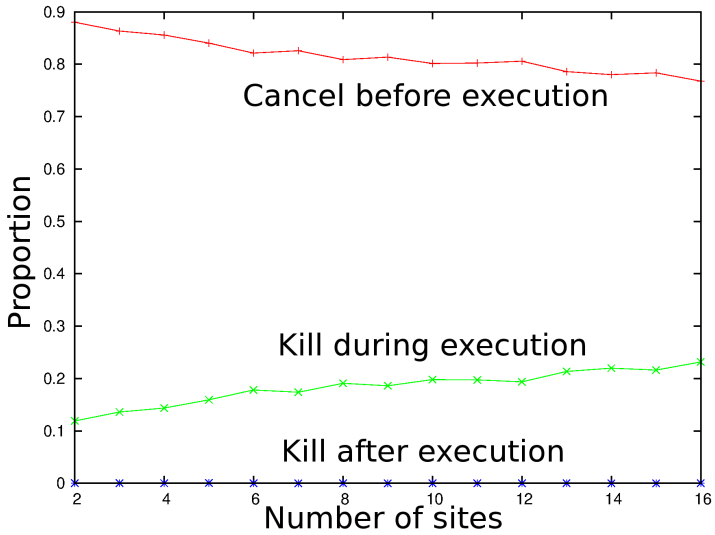


Figure 5.14: Transactions cancelled before, during and after execution (TPC-C, 100 ms message latency)

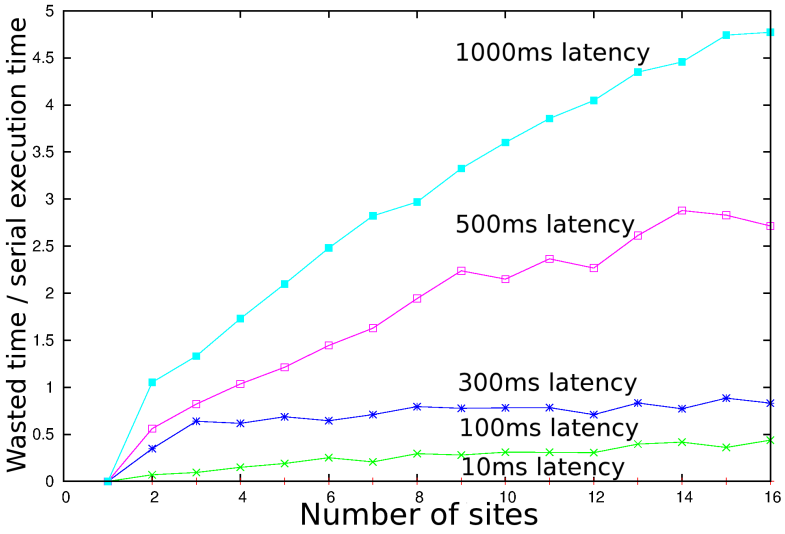


Figure 5.15: Cumulative amount of time spent executing killed and stalled trasactions (TPC-C)

the cost of collision resolution (because the agreement protocol performance depend on message latency). Since the number of workers is unbounded, we expect performance to degrade as the number of sites or their latency increases. We compare the results of TPC-C and TPC-E on Gargamel.

Figure 5.13 shows how speedup varies with cross-site message latencies and number of sites. (Because these simulations are excessively slow, we stop submitting transactions at 3.5 s, i.e., the duration of 512 TPC-C transactions.) Observe that TPC-C exhibits high speedup (close to 16x), thanks to the high degree of parallelism achievable. Parallelism remains high even when inter-site message latency reaches 100 ms, almost twice the cost of a transatlantic message. It degrades seriously only at 5 times the cost of a transatlantic message.

Figure 5.14 takes a closer look at the 100 ms latency, detailing the outcomes of collision. The curves plot the number of transactions that incur a cancel, a kill before ex-

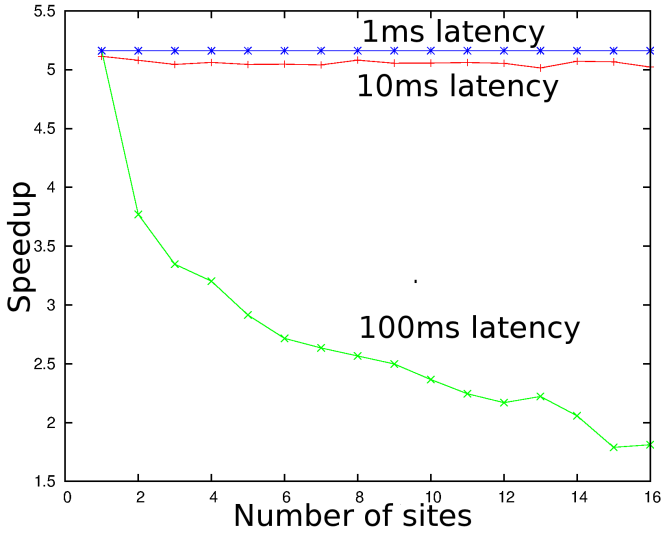


Figure 5.16: Impact of collisions for TPC-E

ecution, and a kill during commit, respectively. Most collisions (more than 70%) result in cancellation before the transaction starts executing. The next figure shows that, as expected, collisions do not affect overall system performance: Figure 5.15 displays the cumulative amount of time wasted executing transactions that will be killed (by definition, cancels do not contribute anything). Note that this figure is almost a mirror image of Figure 5.13: the performance degradation reported therein is roughly proportional to the time wasted by killed transactions.

5.2.4.2 Accuracy Of The Classifier

We performed similar experiments for TPC-E. Figure 5.16 presents speedup for execution of 512 TPC-E transactions, using different message latencies and number of sites. Clearly, speedup is much lower than for TPC-C.

Recall from Section 5.1.2 that the TPC-E conflict estimator conservatively assumes that all pairs of Trade Result transactions conflict; their execution is therefore serialised.

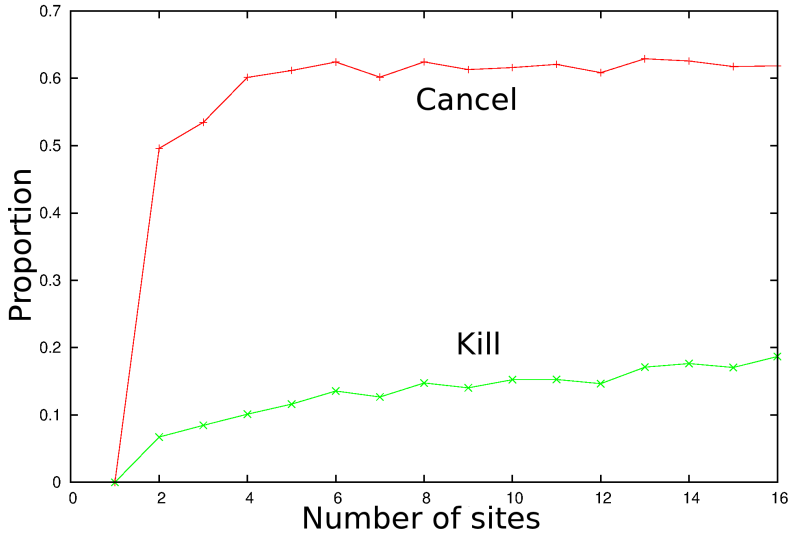


Figure 5.17: Transactions cancelled and killed in TPC-C (100 ms message latency)

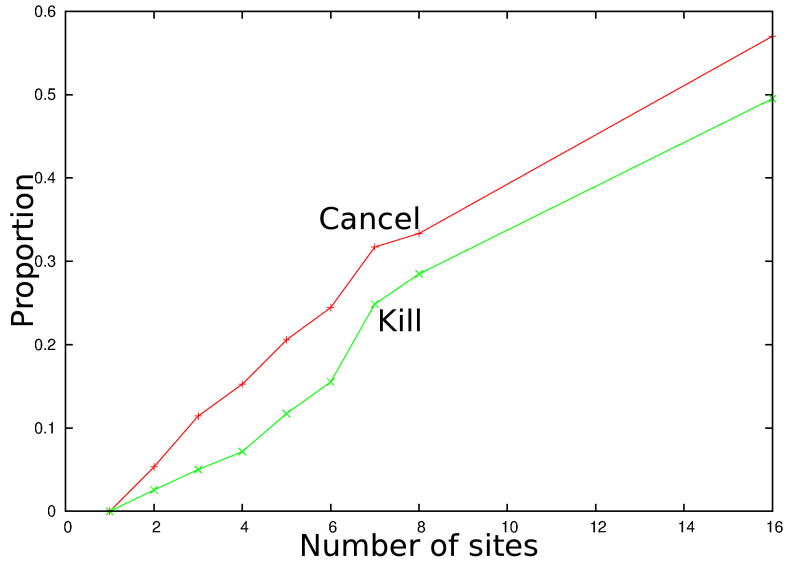


Figure 5.18: Transactions cancelled and killed in TPC-E. (100 ms message latency)

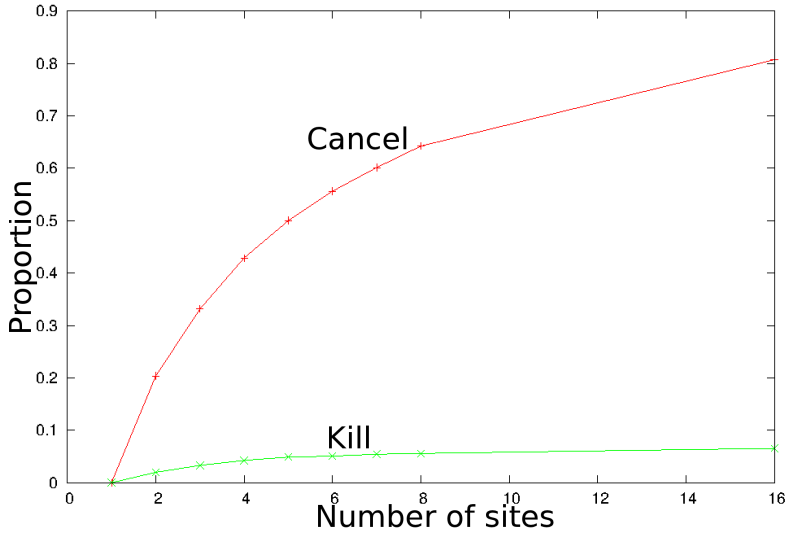


Figure 5.19: Transactions cancelled and killed (TPC-E, 50% of trade result transactions, 100 ms message latency)

Unfortunately, Trade Result transactions dominate the workload; they represent only 10% of the workload in number, but 16,94% in execution time. The theoretical maximum speedup of a system that serialises all Trade Result transactions and paralellises everything else is $100/16.94 = 5.9$.

5.2.4.3 Workload Composition

We further observe that message latency impacts system performance more for TPC-E than TPC-C, because of different numbers of canceled vs. killed transactions.

Consider Figures 5.17 and 5.18, which show the number of transactions cancelled and killed, in TPC-C and TPC-E respectively, assuming a message latency of 100 ms.

In the former, collisions are mostly resolved before the transaction has started executing; for TPC-E, it is the opposite. In fact, the ratio between cancels and kills depends on the ratio between the length of chains and the time to resolve collisions. When it

is high, Gargamel has enough time to cancel a colliding transaction before it starts to execute. When the chains are short relative to consensus latency, transactions start to execute before consensus completes. This happen when transaction duration is low and message latency is high. Indeed, the length of chains depends on the incoming rate and duration of conflicting transactions, and the time to resolve collisions depends on consensus latency.

In TPC-C, performance is dominated by the serialisation of New Order (NO) and Payment (P) transactions, which together represent 88% of transactions. We observe that the time between one incoming NO or P transaction and the next is short enough for Gargamel to be able to cancel and reschedule before the transaction begins to execute. In TPC-E Trade Result are less frequent (only 10% of the workload). The long period between successive Trade Result transactions prevents Gargamel from rescheduling these transactions before they start to execute.

To confirm this intuition, we modify the transaction mix of TPC-E, increasing the proportion of Trade Result transactions to 50%. As Figure 5.19 shows, in this case, the ratio between cancelled and killed transactions is comparable to TPC-C; this modified benchmark does not suffer from message latency, and varying the message latency from 1 ms to 100 ms has only a small effect.

5.2.4.4 Summary

These experiments show the importance of the classifier accuracy and workload composition. In TPC-C, with a good classifier shows a speedup close to 16x. Thanks to the high incoming rate of bottleneck transactions (NO and P, the 88% of the workload) the system does not suffer from kills, even in presence of high message latency.

In contrast, the TPC-E classifier conservatively serialises all Trade Result transactions, resulting in a poor speedup. Moreover, due to the relative low incoming rate of the bottleneck transaction, TPC-E suffers much more than TPC-C from message latency.

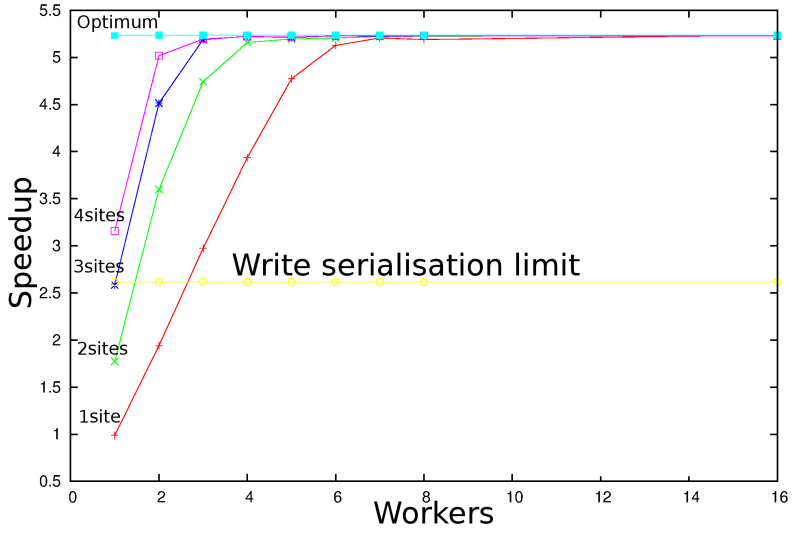


Figure 5.20: Site-worker relationship for TPC-E

5.2.5 Adaptation to The Number Of Workers

We have run several instances of the TPC-E to show how Gargamel performs when varying the number of sites and workers. Figure 5.20 shows the speedup of Gargamel in TPC-E with 1, 2, 3 and 4 sites, while increasing the number of workers per site from 1 to 16. All experiments are performed with a message latency of 1 ms. The speedup quickly reaches an upper limit. When there are about ten workers, the speedup of Gargamel is maximal; adding more resources does not provide any further speedup.

Another interesting point of Figure 5.20 is the relation between sites and workers. One site with two workers performs a little better than two sites with one worker, similarly for two sites with three workers when compared to three sites with two workers. This little performance gain is due to the cost of synchronisation as previously discussed in Section 5.2.4.1.

Using the TPC-E workload, all Trade Result transactions are serialised. Therefore, the speedup of the system depends mainly on the time needed to execute the chain of

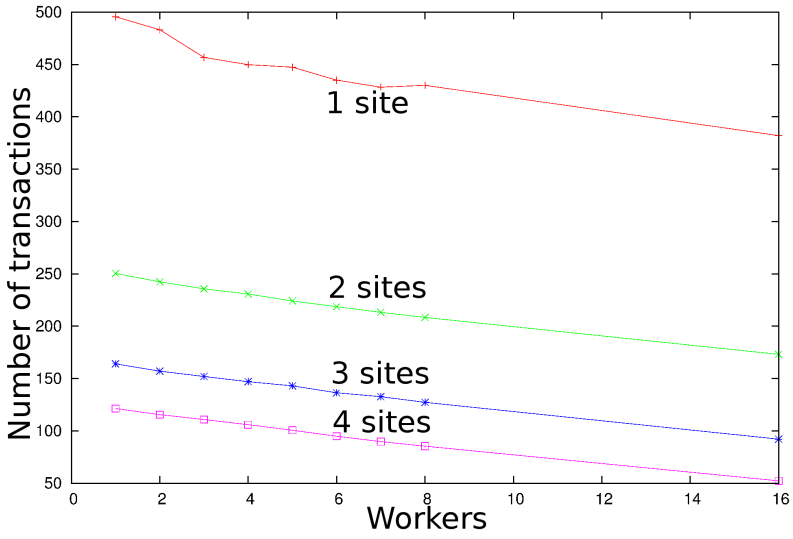


Figure 5.21: Number of transaction that wait for an available worker (TPC-E)

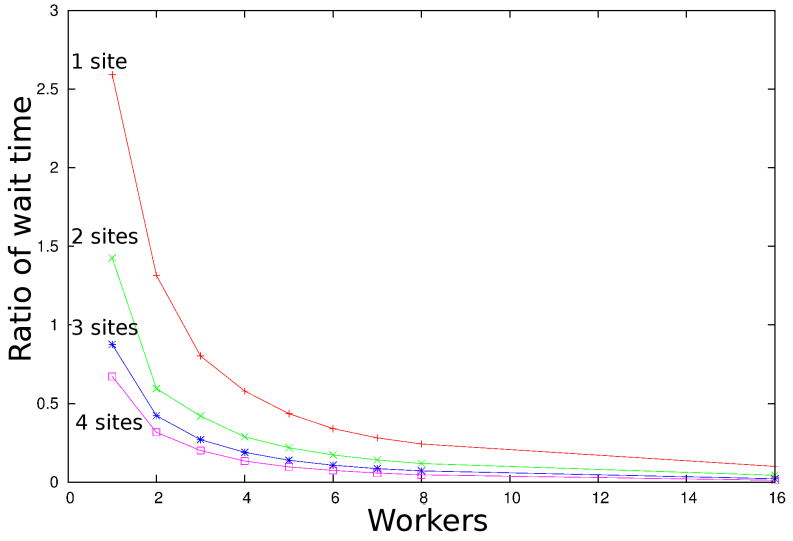


Figure 5.22: Time spent by transactions waiting for an available worker (TPC-E)

Trade Result transactions. To determine the system dimensions for optimal throughput we measured how much time transactions have to wait for an available worker before being executed. Figures 5.21 and 5.22 show the number of transactions that are delayed until a worker becomes available and the ratio of time that transactions spent waiting for available workers. As expected, in Figure 5.21, the number of transactions decreases linearly while increasing the number of workers. In Figure 5.22 the total time spent waiting decreases exponentially with the number of workers. The reason is that increasing the number of workers decreases both the number of waiting transactions and the waiting time.

Our evaluation shows that, TPC-E workload requires substantially more resources to minimize the response time than to reach the maximal speedup (cf. Figure 5.20). This is because of the huge amount of read-only transactions that start execution in parallel at the time they are received by the sites.

5.3 Conclusion

The simulator described in this chapter is useful to quickly experiment different approaches on replication strategies (as primary-copy or update-anywhere replication), concurrency control and isolation levels. We also have tested the feasibility and performance of a transaction classifier for two different benchmarks. The results we obtained by simulations motivated us to implement a full prototype. Several design choices of the prototype are driven by the simulation results presented in this chapter. This is the case for example for the use of the update-anywhere approach and the use of the optimistic synchronization across schedulers in the multi-site setting.

Chapter 6

Gargamel Implementation

Contents

6.1 TPC-C Client Implementation 91

6.2 Scheduler Implementation 91

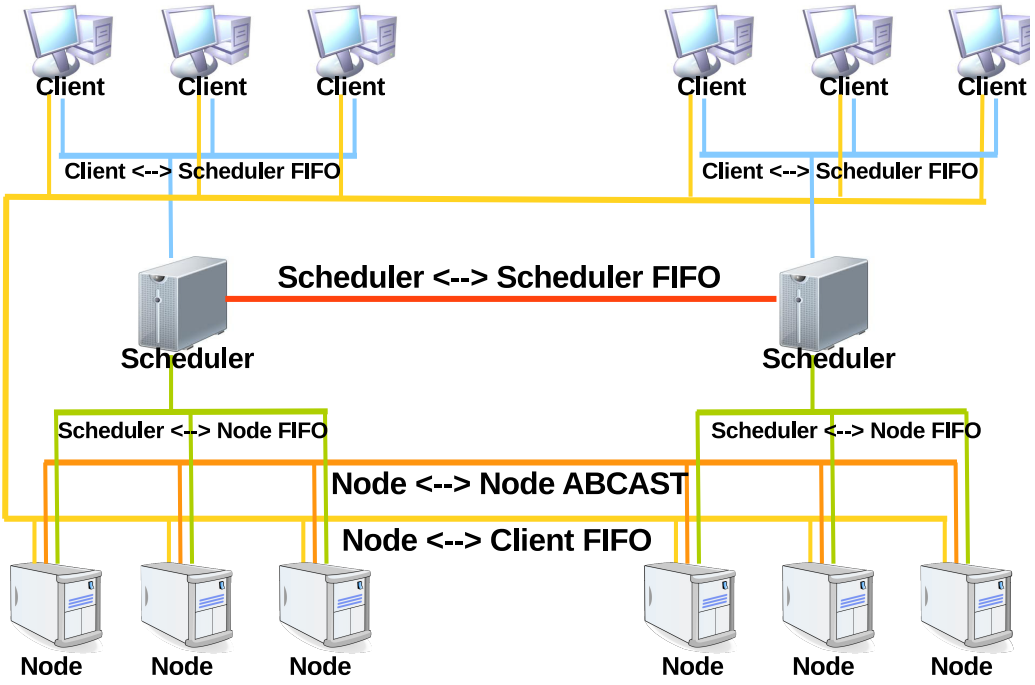
6.3 Node Implementation 92

6.3.1 Certification Algorithm 93

6.4 Classifier Implementation 96

In order to evaluate Gargamel, we built a prototype running on a real database. The purpose of the implementation is to validate the idea in a real setting rather than validate the simulation results.

The main Gargamel prototype components (nodes, scheduler and client emulator) are written in Java (∼12k lines of code). They communicate through JGroups [42], a reliable multicast system used in JBoss [27], an open-source Java application server (now renamed to WildFly). Our concurrency control and update propagation mechanism is based on group communication, and correctness depends on the properties of the communication channels. Figure 6.1 gives an overview of all the communication channels.



Communication channels between nodes, clients and the scheduler.

client <–> scheduler FIFO channel used by clients to send transaction execution requests to the scheduler. The scheduler uses this channel only once to send a "start" message to the client.

scheduler <–> node FIFO channel used by the scheduler to forward transaction execution requests to the selected node. The transaction execution requests message contains also transaction dependencies.

node <–> node ABCAST channel used by nodes to diffuse the write-set of executed transactions and send certification messages.

node <–> client FIFO channel used by nodes to reply to clients once the transaction is committed. Clients never communicate directly with nodes by this channel.

scheduler <–> scheduler FIFO channel used by schedulers to synchronise the transaction graph.

Figure 6.1: Communication channels

There is one ABCAST channel and four FIFO channels. The ABCAST channel link all nodes between them, one FIFO channel links scheduler and nodes, one FIFO channel links scheduler and clients, another FIFO channel links nodes and clients, and the last FIFO channel links schedulers between them.

We use an unmodified version of PostgreSQL [64], an open source relational DBMS for nodes' database. The communication with the database is done through Java Database Connectivity (JDBC).

6.1 TPC-C Client Implementation

A client represent a TPC-C terminal. There is a terminal for each TPC-C warehouse, by default we use 10 warehouses. A terminal generates requests for the scheduler. A request contains procedure call invocations. The parameter generation and the transaction mix respects the TPC-C Standard Specification Revision 5.11 [66].

A clients sends its transaction execution requests to the scheduler, and awaits a reply from the executing node. Clients are unaware of the scheduling strategy, which can be either Gargamel or Round-Robin.

6.2 Scheduler Implementation

A scheduler receives transaction execution requests from clients, and forwards them to the nodes, according to the Gargamel or Round-Robin strategy depending on the experiment. When the scheduler sends a transaction to a node, it specifies which worker should execute the transaction.

When a node executes a transaction, it replies with a commit message directly to the client, without passing through the scheduler. Communication from a node to a scheduler occurs only for garbage collection purposes.

The scheduler implements the scheduling algorithm described in Section 3.3 of Chapter 3. Since New Order transactions can conflict only with other New Order transactions, the scheduler never compares a New Order transaction with a Payment or a Delivery transaction and vice versa. This reduces the number of comparisons while scheduling a transaction by about half. In fact New Order transactions represent $\simeq 49\%$ of the "update" workload, while Payment and Delivery represent together the remaining $\simeq 51\%$.

Communication from client to the scheduler is one-way. The only message sent from the scheduler to clients is a $StartClientMessage$, used at bootstrap time, to start client execution, once all nodes and schedulers are ready.

6.3 Node Implementation

An node is in charge of executing transactions received from the scheduler, and of replying to clients once one of their transaction is committed in the database. If a transaction fails certification, it is aborted in the database, and rescheduled for execution. This is done transparently for the client.

A node manages all communication with its workers; when the scheduler sends a transaction message to the node, it sends also the identifier of the selected worker. The node forwards the transaction to the worker, and the worker adds it to its execution queue. Each worker executes the transactions in the queue serially, in FIFO order. When Gargamel is used, the worker checks, before it executes a transaction, if all its dependencies are satisfied. If so, the transaction is executed right away. Otherwise the worker waits for all its dependencies to be satisfied. Transaction's dependencies are computed by the scheduler, and is sent along with the transaction to the nodes.

Each node contains a full replica of the database. The database is an unmodified version of PostgreSQL with TPCC-UVA [41], an open source TPC-C implementation. The database files reside in a ramdisk: it is fully in-memory. Postgres has been tuned for per-

formance optimisation. We have optimised the maximum number of connections, the cache and the shared buffer size as well as the disk synchronisation (fsync) on commit. Some of those optimisations are functional only when the database resides in disk and not in memory. Database optimisations speed-up transaction execution by an order of magnitude for both Gargamel and Round-Robin.

Round-Robin needs to certify transaction. When certification is used, the node must extract the transaction's write-set, in order to check for write-write conflicts. The communication with the database is done through JDBC, and unfortunately the JDBC interface, and Postgres itself, has no support for write-set extraction. In order to extract the transaction write-set without modifying the DB engine, we have added to the TPC-C stored procedures the necessary logic to collect changes of update transactions in the database and return them as text. Each invocation of a stored procedure returns its updates and inserts, as text. In case of TPC-C, this is very useful, because TPC-C does not contains range queries; given two write-sets we can check for conflict with a simple text comparison on the "where" clause. For other workloads, a simple text comparison might be insufficient and it may be necessary to analyze the update and insert statement text to extract write-write conflicts.

6.3.1 Certification Algorithm

The certification mechanism is implemented as follows:

Gargamel maintains a total order of committed transactions. Each committed transaction is associated with commit number, called its *Global Transaction Sequence Number (GTSN)*. If a transaction commits, it is guaranteed to commit with the same GTSN at all nodes.

Each node maintains a mapping between GTSN and the corresponding transaction write-set.

When a node executes a transaction, it associates to that transaction a temporary

GTSN computed as the last committed GTSN + 1. When execution terminate the node broadcasts, atomically and in total order, a certification request message to all nodes at all sites including itself. The message contains the temporary GTSN and the transaction's write-set. The temporary GTSN represents the *snapshot point* of a transaction, i.e., the number of transactions visible in the snapshot.

When a node receives a certification message for a transaction t, it retrieves the write-set of all committed transactions executed concurrently to t. Transactions executed concurrently to t are all the transactions committed with a GTSNs equal or greater than t's temporary GTSN. The write-set of t is compared with the write-sets of concurrent but committed transactions, to decide whether t is to be aborted or committed. In case, t will be committed with a GTSN equal to the GTSN of the last committed transaction + 1. Since all nodes receive the same certification messages in the same order (thanks to ABCAST) they will commit and abort exactly the same transactions, and committed transactions will be associated with the same GTSN at all nodes.

We now illustrate the commitment algorithm with an example. Consider the simplest case: two nodes, N_1 and N_2, which execute concurrently two transactions, t_1 and t_2. At the begin N_1 and N_2 have the last associated GTSN set to 0 and their mapping $< transaction\ GTSN, writeSet >$ is empty.

N_1 and N_2 receive transactions t_1 and t_2 at the same time. N_1 associates a temporary GTSN equal to 1 (its last associated GTSN + 1) to t_1 and executes it. N_2 also associates 1 as a temporary GTSN to t_2. t_1 and t_2 execute concurrently at N_1 and N_2. When t_1 finish, N_1 broadcasts a certification message $m1 =< 0, writeSet(t_1) >$ containing the transaction temporary GTSN and t_1's write-set. Similarly N_2 broadcasts $m2 =< 0, writeSet(t_2) >$. Say that ABCAST delivers first $m1$, then $m2$ at all nodes.

Both N_1 and N_2 receive the $m1 =< 0, writeSet(t_1) >$ certification message and check in their map for transactions committed with a GTSN equal or greater than the temporary GTSN of t_1. Since the $< transaction\ GTSN, writeSet >$ map is empty they do not found concurrent transactions and they both commit t_1 with GTSN equal to 1 (the last

Transaction pairs	Conflict condition
$NO(w_1, d_1, I_1) \times NO(w_2, d_2, I_2)$	$(w_1 = w_2 \wedge d_1 = d_2) \vee I_1 \cap I_2 \neq \emptyset$
$P(w_1, c_1, cw_1, cd_1) \times P(w_2, c_2, cw_2, cd_2)$	$(w_1 = w_2) \vee ((cw_1 = cw_2 \wedge cd_1 = cd_2) \wedge (c_1 = c_2))$
$D(w_1) \times D(w_2)$	$w_1 = w_2$
$D(w_1) \times P(w_2, c_2, cw_2, cd_2)$	$w_1 = cw_2$

The subscripts represent two concurrent transactions. Please refer to Section 6.4 for an explanation of variable names.

Table 6.1: Conflicts of TPC-C under Snapshot Isolation

GTSN +1). They update the $< transaction\ GTSN, writeSet >$ map with the new value $< 1, writeSet(t_1) >$.

When N_1 and N_2 receive $m2 =< 0, writeSet(t_2) >$ they both find that t_1 committed with a GTSN (1) equal than t_2's temporary GTSN. They check for an intersection between t_2's write-set (sent in the $m2$ message) and t_1's write-set (stored in the local map), and both deterministically abort t_2 if t_1 and t_2 write-sets intersect, and or both deterministically commit t_2 with $GTSN = 2$, if not.

This algorithm relies on the ABCAST primitive for its correctness. The algorithm imposes a total order broadcast Once a message is delivered, each node deterministically commits or aborts without any further communication. The price to pay to the one-step committment is that each site stores the write-set of all committed transactions. We periodically (and synchronously) garbage-collect entries in the $< transaction\ GTSN, writeSet >$ map. A variant of this algorithm is to not store the write-set of remotely executed transactions and to exchange votes on the certification outcome. If all sites vote for commit, the transaction is committed; if at least one site votes for abort the transaction is aborted. We have chosen to implement the one-step certification protocol because i) in our workload transaction write-sets are small and ii) in a geo-replicated setting latency, message loss and partition probability can make the vote exchange for agree on the certification outcome inefficient.

6.4 Classifier Implementation

The classifier is a key component of Gargamel. It allows Gargamel to predict conflicts in order to avoid certification and aborts.

The classifier of our prototype is based on a static analysis of the TPC-C benchmark. We have analyzed the access pattern of each transaction type to understand whether two concurrent transactions can potentially write on the same field or not. If so, they are considered conflicting. We consider only write-write conflicts, because we currently support only the SI and the PSI isolation level. However, the static analysis can be easily extended to consider read-write conflicts.

The classifier TPC-C-specific implementation is the same used for the discrete event simulator and is described in Section 5.1.1.

Chapter 7

Gargamel Evaluation

Contents

7.1 Single Site Evaluation 98

7.1.1 Benefits of the Pessimistic Approach 99

7.1.2 Benefits of the Passive Replication Approach 104

7.1.3 Resource Utilisation 104

7.2 Multi-Site Evaluation 107

7.2.1 Benefits of the Multi-Site Deployment 109

7.2.2 Impact of Database Performance 110

7.2.3 Impact of collisions 113

7.3 Conclusion 115

It this Chapter we compare Gargamel against a simple Round-Robin scheduler. We omit the comparison with Tashkent+ because in all our experiments with the database on disk the memory used by nodes is below the available memory. For other experiments we used an in-memory database. This make useless, in our settings, to compare with a system that optimise the memory usage as Tashkent+ does.

Default parameters	**Value**
Number of nodes	64
Number of workers per node	1
Incoming rate	50/100/150/200 t/s
Load (single-site)	100.000 transactions
Warehouses (TPC-C)	10

Table 7.1: Experiment parameters

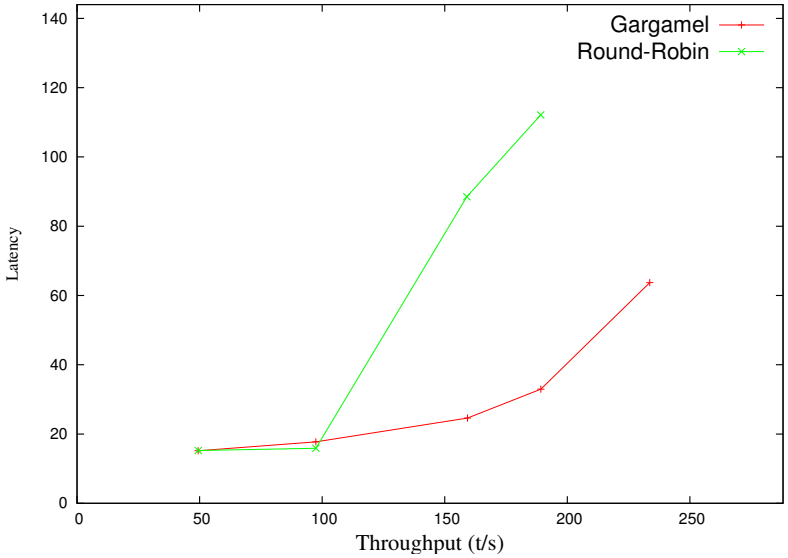


Figure 7.1: Impact of aborts

7.1 Single Site Evaluation

In this set of experiments we evaluate Gargamel in EC2. We vary the input incoming rate and we measure the client-perceived response time (i.e., time elapsed between the time the client submits a transaction, and the time it receives the reply) and throughput (i.e., the number of transactions per second committed by the system). Unless differently specified, we use the parameters from Table 7.1.

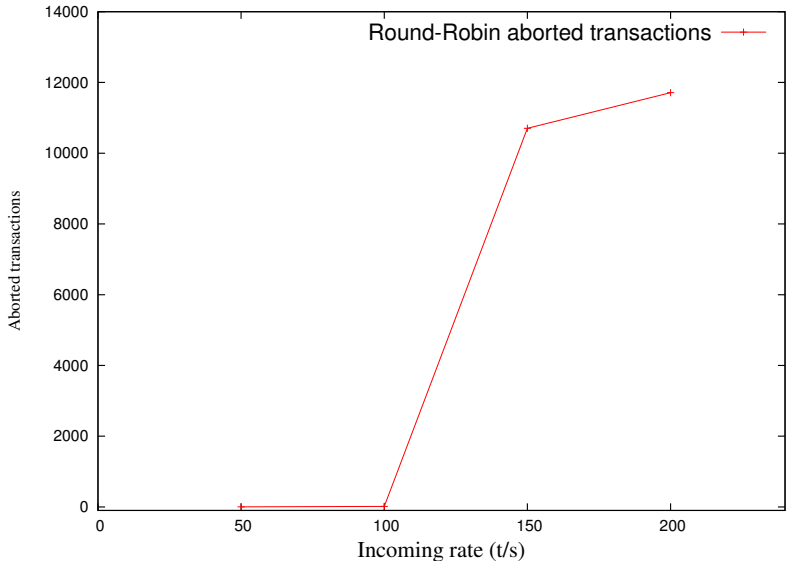


Figure 7.2: Impact of aborts (2)

7.1.1 Benefits of the Pessimistic Approach

Figure 7.1 compares Gargamel versus a Round-Robin scheduler. In order to show the impact of aborts at different incoming rates, in this experiment, both systems certify transactions, even if Gargamel does not need to certify TPC-C transactions. Since both systems use the same certifier we can see that i) the Gragamel overhead at low incoming rate and ii) the impact of aborts in a Round-Robin approach, when incoming rate (and consequently contention) increases.

As shown in Figure 7.1, when contention is low (i.e., incoming rate under 100 transactions per second) Gargamel and Round-Robin have similar performance. In this situation, the system is under-loaded, and any scheduler would have similar performance. At 100 transactions per second Gargamel response time is sightly higher than in the Round-Robin case because of scheduling overhead. As contention increases, Round-Robin increases the number of transactions aborted due to negative certification, causing the latency to increase. In the other hand, Gargamel does not suffer from aborts, and

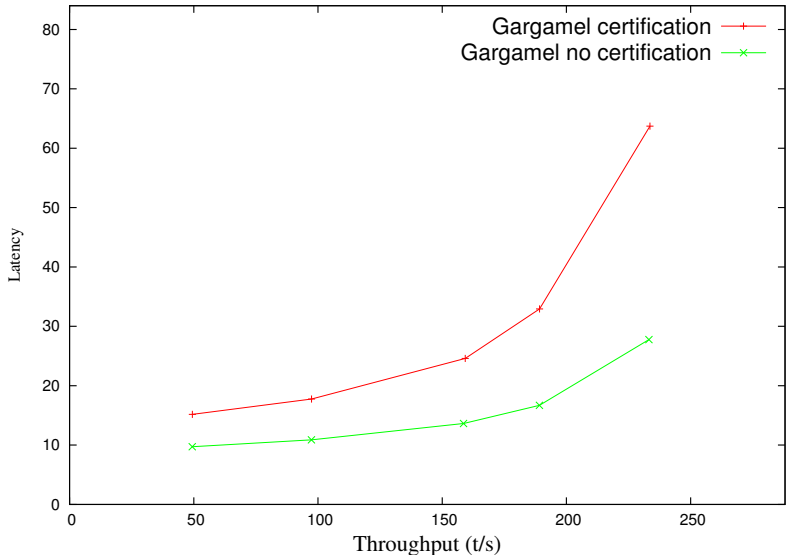


Figure 7.3: Impact of certification

has lover latency and scales better.

This explanation is confirmed by Figure 7.2, showing the abort rate of the Round-Robin scheduler in the same execution as Figure 7.1. The plot in Figure 7.2 has the same shape as the latency: a very low abort rate until the incoming rate reaches 100 transactions per second, an explosion between 100 and 150 transactions for seconds, then both plots flatten again. This explains that the latency increase of Figure 7.1 is determined by the aborts. Indeed, each time a transaction is aborted, it has to be executed and certified again, before being able to reply to the client. This has a negative impact on both the load of the system and the client-perceived latency.

Even if Gargamel does not needs certification, to study the impact of certification, we run Gargamel with and without certifying transactions. As showed in Figure 7.3 when Gargamel runs without certifying transactions it has better performance (lower latency for clients) and better scalability (higher throughput at low latency). Certification prevents the system from scale more than 200 transactions per second, without greatly increasing the client perceived latency. When the system is saturated, both the

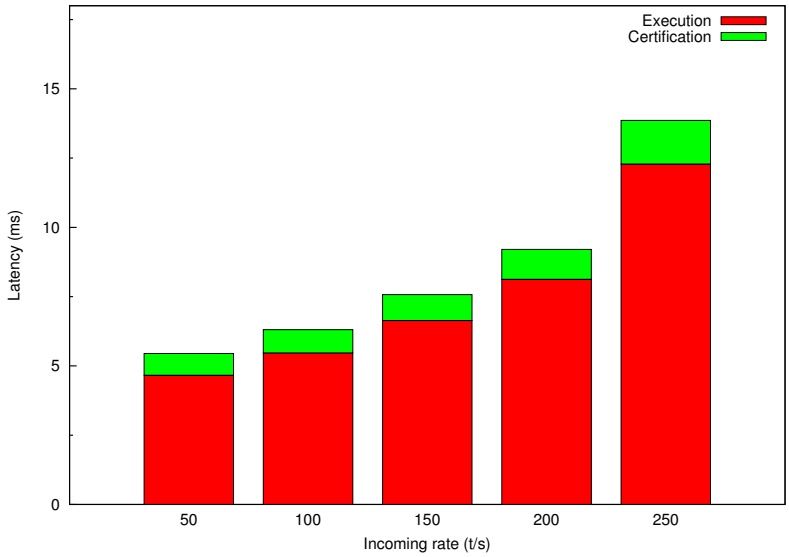


Figure 7.4: Execution / certification latency

transaction execution and the certification time increase. Figure 7.4 compares the time spent running transactions in the database to the time spent certifying.

Figure 7.5 shows, for each TPC-C transaction type, how long it takes for Gargamel to schedule it, varying the incoming rate. Order Status and Stock Level transactions take very little time (less than 5 microseconds) regardless of incoming rate. This is because they are read-only transactions and they do not need to be scheduled (i.e., we do not need to traverse the graph searching for conflicts), so they are assigned to a free worker as soon as they come.

Payment and Delivery, which are read-write transactions, have higher latency, which increases slightly as the incoming rate increases, because they must be checked against already-scheduled Payment and Delivery transactions. Scheduling is fast (less than 150 μs) because of the simplicity of the check at the classifier and the high conflict rate. Indeed, the chains check stops as soon as a conflict is found: it is useless to check for conflicts in the ancestors of a conflicting transaction. For details see the scheduling algorithm description in Section 3.3.

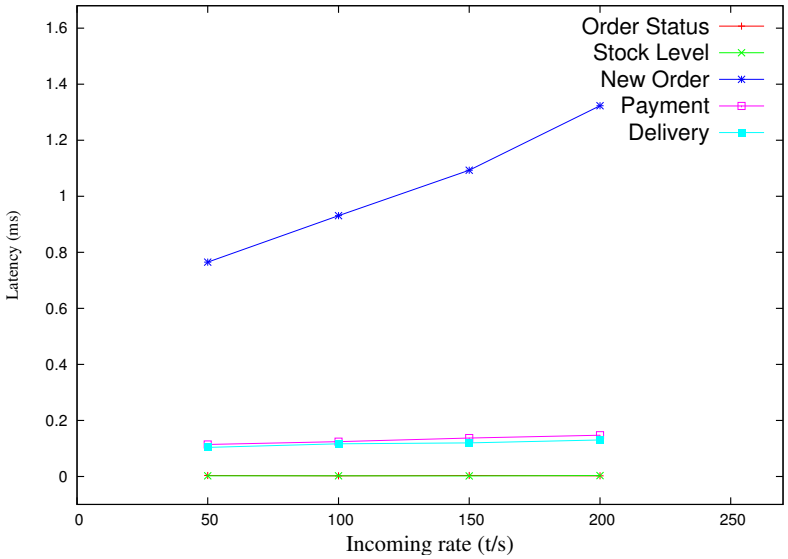


Figure 7.5: Scheduling time

New Order transactions take longer to be scheduled (between 800 and 1.300 μs) and incoming rate has more impact on scheduling performance. This is due to the combination of two effects: first the classifier is more complex (it must check for intersections between $Items$ sets), and second as acrlongno have lower conflict rate, an incoming transaction is more likely to check many or all the already scheduled New Order transactions. However, in all the cases, scheduling cost remains an order of magnitude lower the certification cost.

Figure 7.6 shows the comparison +between Gargamel and Round-Robin in their default settings, i.e., Gragamel does not certifies transactions and Round-Robin does. The Gargamel and Round-Robin plots are the same as in Figure 7.3 and Figure 7.1 respectively. When the system executes more than 100 transactions per second, it starts to suffer from contention. The certification-based approach, required by Round-Robin, suffers from high latency and poor scalability due to the cost of certification and the abort rate. Gargamel is able to scale much better and it keeps the client perceived latency low.

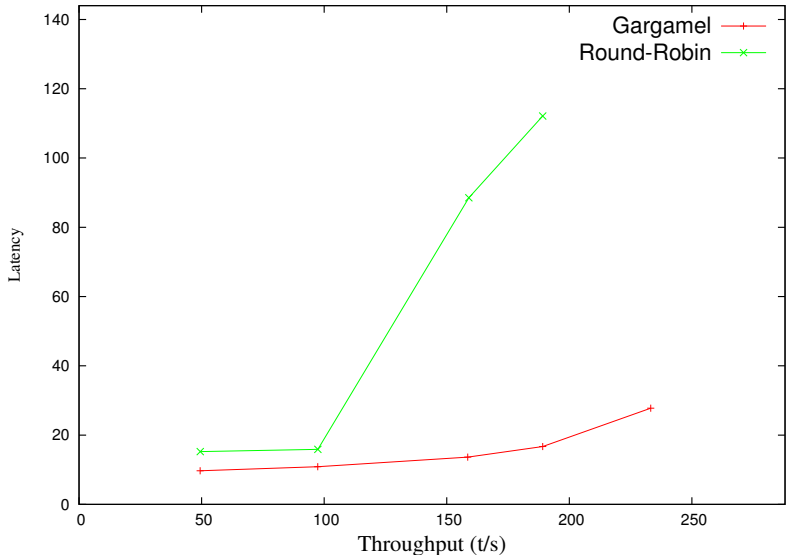


Figure 7.6: Impact of certification (2)

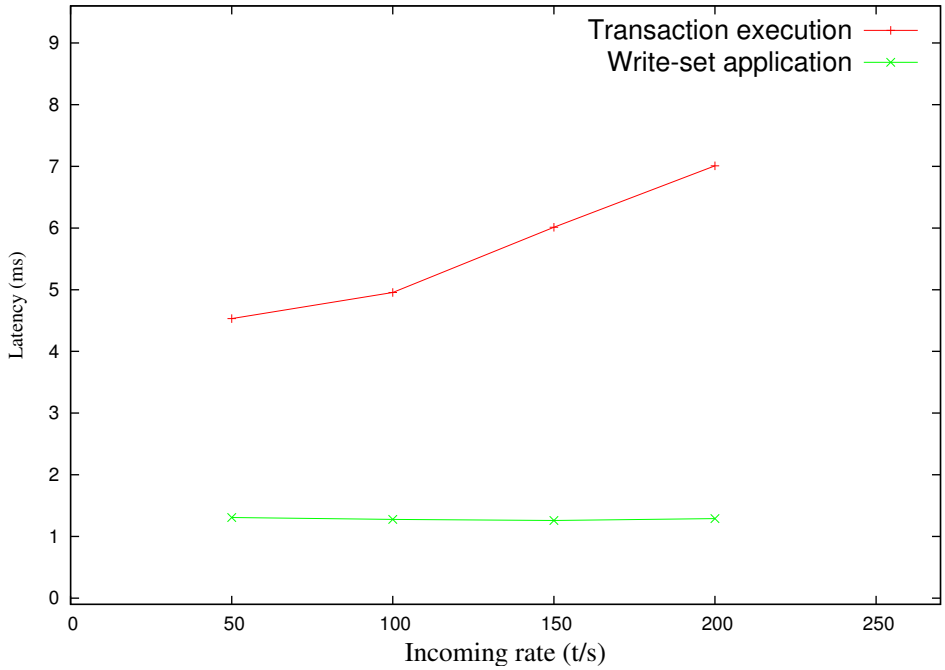


Figure 7.7: Transaction execution time VS write-set propagation

7.1.2 Benefits of the Passive Replication Approach

Figure 7.7 shows the average database latency to execute a transaction and to reflect the write-set as the incoming rate varies. The difference of latency between the execution of a transaction and the application of its write-set has an important impact because allows to improve scalability of full-replication.

Since each replica has to apply the write-set of all the transactions, or to execute all of them, if applying the write-set in a database is more efficient than re-executing the full transaction, then a set of database replicas can increase the update throughput than a single replica achieve.

In TPC-C, transactions have a heavy footprint compared to updated records (i.e., they access lot of records and tables and update few of them). As shown in Figure 7.7, this makes applying the write-set four to seven times faster than executing the transaction. Moreover, Figure 7.7 shows that the time to apply the write-set is constant, i.e., does not depends on the incoming rate. At the opposite, transaction execution time increase as the incoming rate increases as the DB load augment.

7.1.3 Resource Utilisation

We now examine the resource utilisation and the size of the wait queues at "runtime", during the execution.

The bottom part of Figure 7.8 shows the number of busy nodes as the execution advances in time. The top part shows the number of transactions waiting for a reply at clients (note that for readability the scale of the y axis differs between the bottom and the top). Vertical lines represents the time at which transactions stop arriving and the incoming rate goes to zero.

For 64 workers, at high incoming rate (600 transactions for second), Gargamel uses all the nodes; on the client side, queue size reaches about 50000 transactions. Round-Robin uses all nodes as well, but there client side the queue size is higher, at around

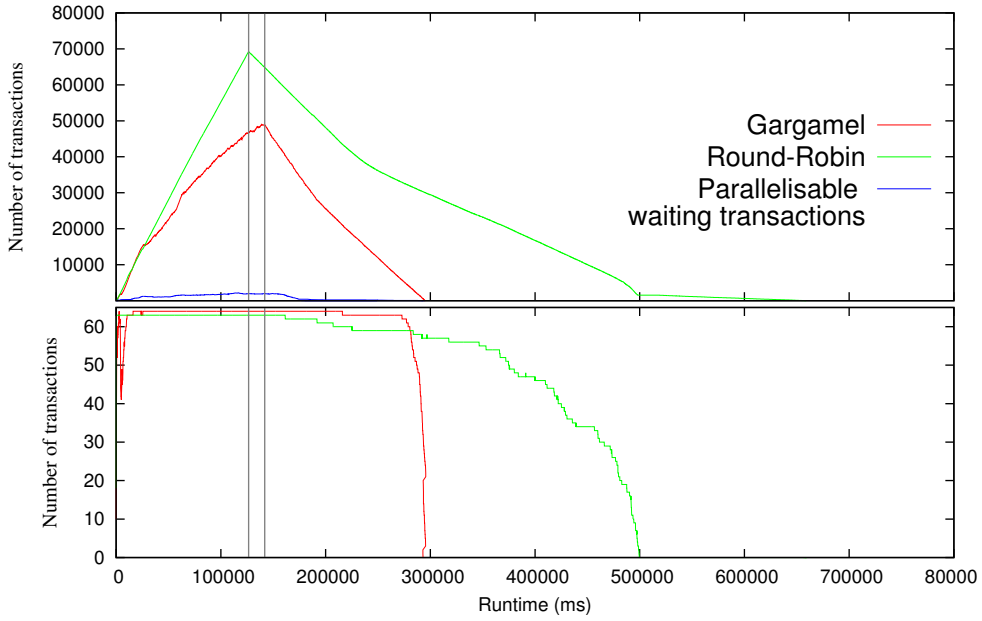


Figure 7.8: Resource utilisation

70000 transactions. This is due to the fact that some proportion of the transactions executed by Round-Robin are aborted, and the client is not receiving replies for those transactions until they are re-executed and committed. We recall that this is not the case for Gargamel, because it does not abort transactions.

As showed by the "Parallelisable waiting transactions" curve of Figure 7.8, the number of paralleliable transactions (i.e., transactions that can execute in parallel without conflicting) gets higher than the number of nodes. In fact there is a relatively small number of transactions in the nodes waiting queue that can be parallelised, this means that Gargamel would benefit from some extra nodes. However, the large majority of waiting transactions are transactions that are postponed in Gargamel's chains because of conflicts, so it would be useless to parallelise them.

In contrast, the policy of Round-Robin is to execute as many transactions in parallel as possible, as soon as they arrive. However, since many of those transactions conflict, there are many aborts, and they do not make progress. They quickly saturate the number of workers; incoming transactions are delayed, and wait queue size grows rapidly.

In the second phase, after transactions stop arriving (represented by vertical lines), Gargamel, according to simulations results (see Section 5.2.2), is expected to free most of the workers. Indeed, at this point, all the read-only and non-conflicting transactions have finished executing; Gargamel only needs a few workers for the remaining chains of conflicting transactions. However, contrary to expectations, Gargamel uses all its workers until the end of execution, then frees all the nodes at almost the same time. We are investigating this discrepancy between simulation and implementation results to figure out if it is just the effect of an implementation or measurement problem, or there is some convoy effect due to the Gargamel design.

Round-Robin, in the second phase, continues to saturate the workers by attempting to parallelise conflicting transactions.

At some point during the second phase, after approximately 270 seconds, Gargamel and Round-Robin continue to empty the queue of waiting transactions more slowly.

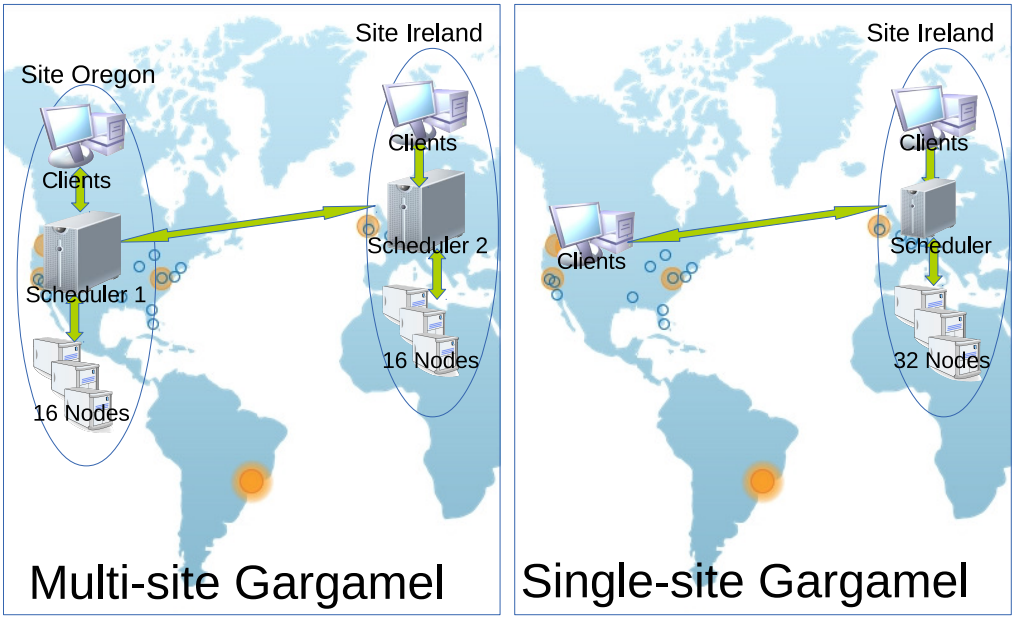


Figure 7.9: Single site and multi-site Gargamel deployment

This is because, at this point, all the read-only and non-conflicting transactions have terminated, and the conflict rate increases. In fact, after this point, there are no more parallelisable waiting transactions. This causes Round-Robin to abort more, and Gragamel to be more likely to wait for dependencies before run a transaction. In both cases, the result is a decrease of throughput that causes emptying the waiting queue more slowly. Gargamel is less affected by the throughput decrease, and performs better than Round-Robin in terms of throughput and resource utilisation.

7.2 Multi-Site Evaluation

In this section we evaluate multi-site Gargamel. In our experiments we measure the client-perceived latency and the impact of collisions while varying the incoming rate.

Amazon EC2 limits the maximum number of instances in a region. The default limit

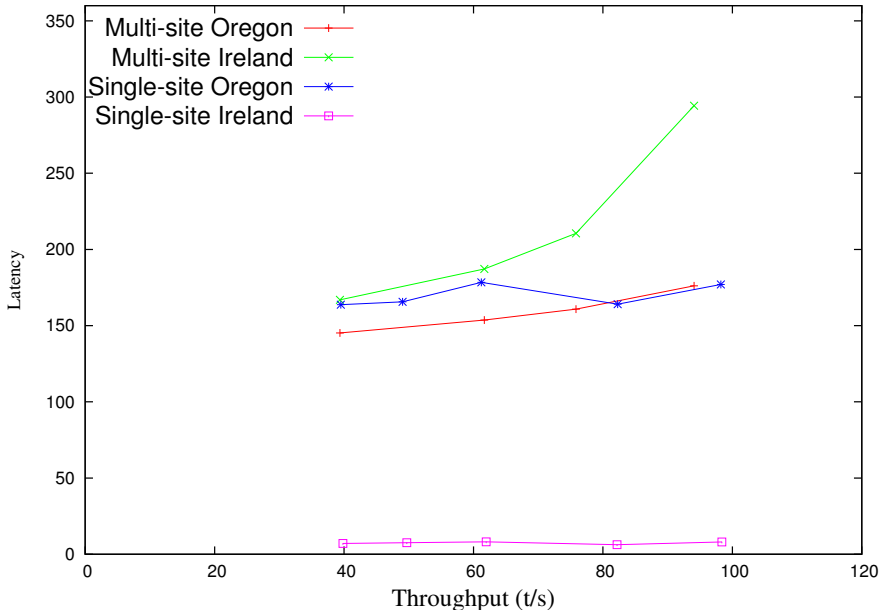


Figure 7.10: single site VS multi-site Gargamel

is 20 machines. In order to run the previous experiments with 64 nodes we have extended this limit to 100 machines for the Ireland EC2 region. In this set of experiments we use 32 nodes (instead of 64 as in single site experiments) to be able to deploy multi-site Gargamel with 16 nodes in two EC2 regions without the need to extend the default maximum instances limit. Notice that the smaller number of nodes compared to previous experiments is compensated by a lower incoming rate. While in previous experiments we vary the incoming rate from 50 to 250 transactions per second, in this set of experiments we vary the incoming rate from 20 to 100 transactions per second. If not specified otherwise, each point in the following plots is taken from a single experience. An experience consists in the execution of 100k transactions. Figure 7.9 shows single site and multi-site settings used for the following experiments. In single site setting, we deploy 32 nodes, a scheduler and a client in Ireland EC2 region and a client in Oregon EC2 region. In multi-site setting we deploy 16 nodes, a scheduler and a client in both Ireland and Oregon EC2 regions.

7.2.1 Benefits of the Multi-Site Deployment

In this set of experiments we compare multi-site Gargamel against single site Gargamel using the deployment described in the previous section and illustrated by Figure 7.9. We measure the client-perceived latency (i.e. the time elapsed between the time the client sends the transaction and the time it receives the corresponding commit message). Figure 7.10 shows the latency perceived by clients in Ireland and Oregon in the single site and multi-site configuration varying the incoming rate. The latency perceived by the client in Ireland on the single site setting is an order of magnitude lower than the latency experienced by all other clients. This is because in single site Gargamel transactions coming from the Ireland EC2 region are executed in the local datacenter and do not need to synchronise remotely. In the other hand, transactions coming from clients located in Oregon in the single site Gargamel case show a much higher latency because they suffer for the high latency between the west of Europe and the west of the United States. Interestingly, the latencies observed for the client located in Oregon in the seingle site configuration, are similar to the ones observed in the multi-site configuration, a little bit higher than multi site Gargamel clients in Oregon and a little bit lower than multi-site Gargamel clients in Ireland. This is because from whatever client multi-site Gargamel receive a transaction, it synchronises with the other scheduler, paying the price of a transatlantic round-trip message. In multi-site Gargamel, clients in Oregon have a lower latency than clients in Ireland because multi-site Gargamel elects a scheduler do be the leader of the agreement protocol. The leader resolves conflicts in case of collisions. In our experiments the leader is the Oregon scheduler, giving a small advantage to clients connected to that scheduler.

In average, multi-site Gargamel does not show any improvement or overhead over single-site Gargamel for transactions that come from distant clients. This is because at low incoming rate, as in this experiment configuration, transactions are executed as soon as they arrive, so multi site Gargamel does not form chains and the agreement latency cannot overlap the time the transaction spend waiting for the execution of pre-

Default parameters	**Value**
Number of nodes	32
Number of workers per node	1
Incoming rate	20/30/40/50/60/70/80/90/100 t/s
Load (single-site)	10.000 transactions
Warehouses (TPC-C)	10

Table 7.2: Experiment parameters for in-disk database

vious transactions in its chain. The optimistic scheduling benefit comes from the fact that the agreement on the transaction execution order between sites proceeds in parallel with the transaction execution. So it is effective only if a transaction has a long execution time or if it waits for previous transactions in the chain to be executed, in such a way that when the transaction starts to execute, the system has already reached the agreement on its position in the graph.

7.2.2 Impact of Database Performance

We evaluate the impact of transaction execution latency running the same experiments of Section 7.2.1 in a less performant database. The next set of experiments is based on a in-disk PostgreSQL database not tuned. Points in the following plots are produced with a single run of 10k transactions. We reduced the number of transactions of each experiments from 100k to 10k to keep constant the total "running time" i.e. the time it takes for the system to execute all the transactions. Unless differently specified, we use the parameters from Table 7.2.

The in-disk database is more than ten times slower than the in-memory database. Figure 7.12 compares the execution time of write TPC-C transactions in the in-disk and in-memory tuned database.

In the experiments presented by Figure 7.12 we compare how multi-site and single

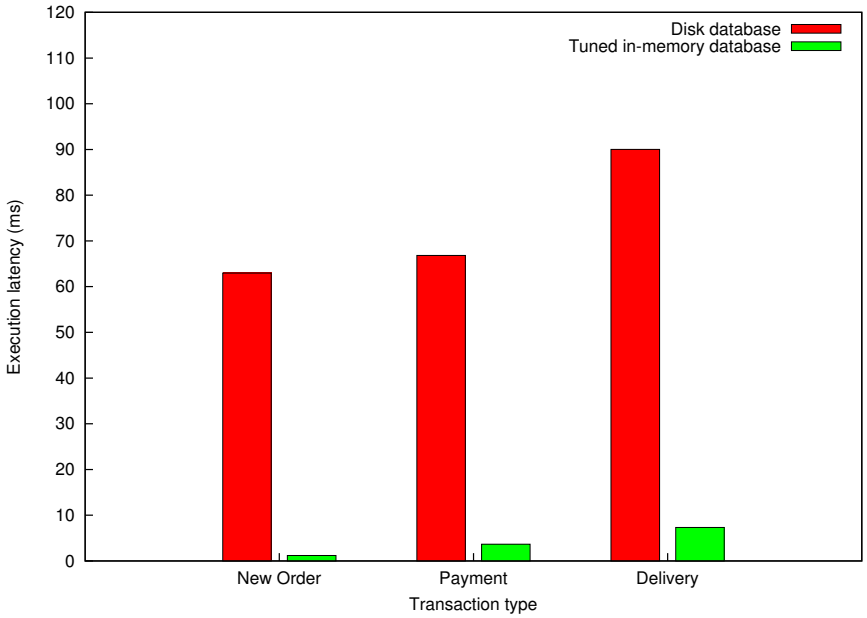


Figure 7.11: Transaction execution time of in-disk and in-memory database

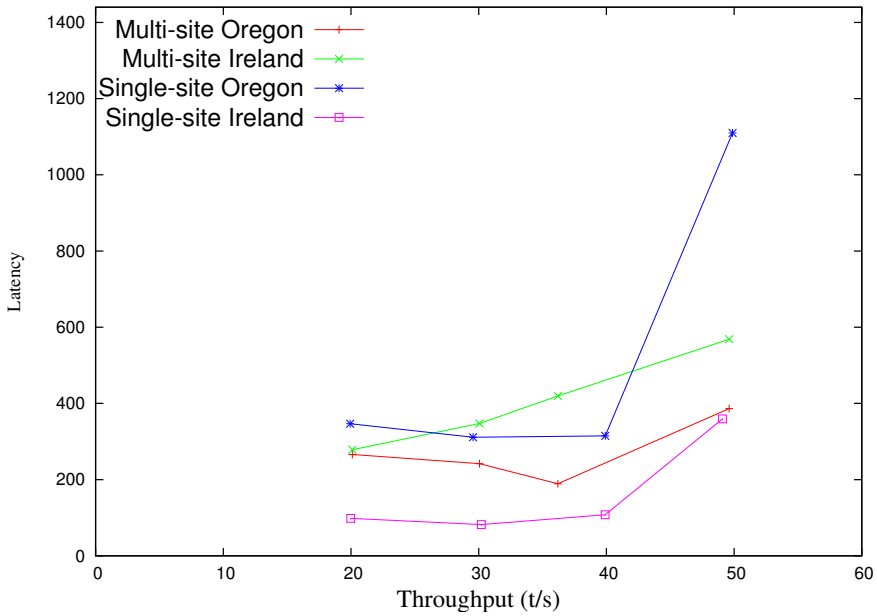


Figure 7.12: Single site Gargamel VS multi-site Gargamel slow database

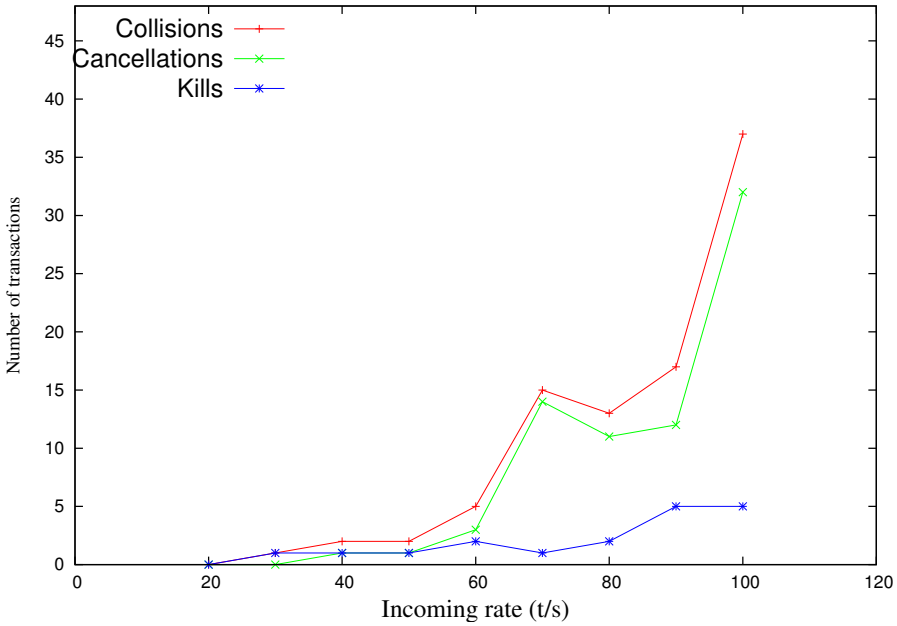


Figure 7.13: Impact of collisions

site Gargamel perform w hen using a slow in-disk database. As for the experiments illustrated by Figure 7.10, we measure the latency perceived by different clients located in different regions as the incoming rate, and consequently the throughput, increases. The main difference with executions using an in-memory database is that even the latency of clients in Ireland in single site Gargamel experience a high latency, and the difference with other clients is smaller than for the in-memory database. This is due to the long transaction execution time (between 50 and 90 millisecond). In this experiment the incoming rate is low, so transactions are executed as soon as they arrive in the system. They do not form chains and the higher execution time is not enough to show the benefits of the optimistic scheduling approach.

7.2.3 Impact of collisions

The optimistic scheduling is interesting when Gargamel forms transactions chains. To show this, we have to saturate the system. At high incoming rates Gargamel organises transactions in chains according to their conflict relations. Multi-site Gargamel schedules transactions at each site according to the local view, then synchronises schedulers optimistically. In this way multi-site Gargamel masks the agreement delay (because the agreement is performed in parallel with the execution of the transactions in the chain), but collisions on the transaction execution order at different sites are possible. In order to evaluate the impact of collisions and the effectiveness of the optimistic approach we saturate Gargamel schedulers pushing the incoming rate up to 100 transactions per second in the in-disk, slow, database.

Figure 7.13 shows the number of transactions that collide, and among colliding transactions how many are cancelled and how many are killed. We recall that a transaction is cancelled when it is rescheduled before its execution starts, and is killed when its execution is started and have to be aborted in the database. The key difference is that cancellations do not cause waste of work because the transaction is not executed at any replica but just rescheduled at a different position while kills involve executing and aborting a transaction at a node, wasting resources. As showed in Figure 7.13, when the incoming rate is low, at 30 transactions per second, there are few collisions. Those collisions result in a kill of the corresponding transaction, because transactions are executed as soon as they arrive and there is no time to reschedule them. Until 60 transactions per second, collisions are very rare and roughly half of them cause the transaction to abort (kill) and the other half cause a simple rescheduling (cancellation). When the incoming rate increases, the number of collision increases as well and the effectiveness of the optimistic protocol became visible: most of the collisions cause a simple cancellation and transactions are rescheduled before being executed. At an incoming rate of 80 transactions per second and more, the number of collision increases rapidly (from 10 to more than 30), but the number of collisions that cause lost work, remains stable and low. Even

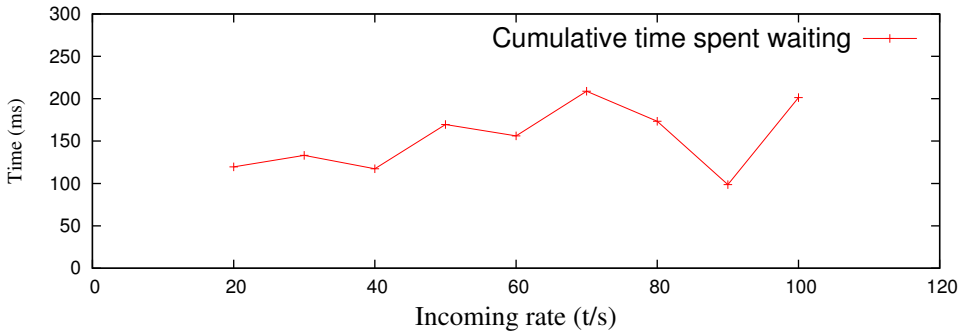


Figure 7.14: Cumulative time spent waiting for agreement

if multi site Gargamel experience some amount of collisions, the lost work remains low, and the degradation of the resource consumption optimality is modest.

The optimistic scheduling imposes to synchronise schedulers to agree on the transaction position in the graph. Transactions cannot be committed in the database until the agreement on their position in the chain is reached. If a transaction execution finishes before the schedulers have agreed on its position, the transaction wait for the agreement process to finish before being commit or killed. In the next experiment we have measured, for increasing incoming rates, the cumulative time spent by transactions waiting for the outcome of the agreement. Figure 7.14 shows the cumulative time spent by transaction waiting for the outcome of the agreement as the incoming rate increases. The cumulative waiting time is between 100 and 200 milliseconds for all the runs. The workload of a run is composed by 10k transactions, consequently in average the transaction commitment is delayed by a time between 0.001 and 0.002 milliseconds. Considering that in this setting (slow in-disk database) the average transaction execution time is between 60 and 90 milliseconds, the delay on the commit time is negligible. The delay is so small because most of the transactions do not wait at all: at the time the transaction is executed the agreement protocol is done and the transaction can commit right away; and as showed before, cases in which the transaction cannot commit and should be kill are rare.

7.3 Conclusion

In this chapter, based on experimentations, we show that Gargamel is able to scale better and with lower client-perceived latency than a Round-Robin certification-based approach. This performance improvement is due to the pessimistic concurrency control that has the advantage to avoid both aborts and certification.

The prototype roughly confirms our simulation results concerning the system latency and scalability. However, for the resource utilisation the simulation results differ; we are actually investigating this discrepancy.

We also show the benefits of the passive replication approach in the TPC-C benchmark where executing a transaction is much slower than applying its write-set.

The comparison between single site and multi-site settings shows that in terms of latency, at low incoming rates, multi-site Gargamel does not introduce any overhead but does not provide any substantial benefits over a single site deployment in terms of performance. Replicating schedulers in distant sites cannot improve latency: even if clients-scheduler latencies are low, to achieve strong consistency, distant schedulers must synchronise, paying the price of high-latency WAN connections. However, multi-site Gargamel does not introduce any overhead for distant clients thanks to the optimistic synchronisation. The ability to replicate at distant sites is suitable for disaster tolerance and permits to expand the system beyond a single datacenter capacity. We show that at high incoming rates collisions augment, but they are mostly resolved just rescheduling transactions and do not cause wasted work. Multi-site Gargamel can impose some delay on transaction commit because of agreement, we show that that delay is negligible.

Chapter 8

Conclusion

Contents

8.1 Summary 117
8.2 Perspectives 119

8.1 Summary

Our approach, Gargamel, is a middelware that maximizes parallelism in distributed databases while avoiding wasted work. Instead of parallelising all transactions, including conflicting ones like most previous systems, Gargamel pre-serialises conflicting transactions. The key idea is to check for conflicts before transaction execution and to order them according to a transaction dependency graph. Transactions are scheduled in such a way that conflicting transactions are never executed concurrently, and non-conflicting ones are parallelised. The advantage of this approach is threefold:

Maximize parallelism: Non-conflicting transactions are spread out to different replicas for parallel execution.

Avoiding wasted work: Transactions are never aborted and re-executed.

Minimizing synchronisation: Transaction execution is conflict-free, so there is no need for certification.

This improves average response time, throughput and scalability, while decreasing resource utilisation, compared to a certification-based system.

Accuracy of the classifier impacts both the amount of parallelism and wasted work. If the classifier suffers from false positives, parallelism can be sub-optimal because non-conflicting transactions are unnecessarily serialised. If it suffers from false negatives, resource utilisation is sub-optimal, because transactions can abort. Additionally, with false negatives, Gargamel cannot avoid certification, which is costly.

Our proof-of-concept classifier uses a simple static analysis of the workload. We exhibit a classifier for the TPC-C benchmark, and we illustrate how to avoid false negatives and to limit false positives. We also discuss the effects of an imperfect classifier.

In a multi-site setting, Gargamel supports multiple schedulers which proceed optimistically in parallel, possibly resulting in collisions. However, our simulations shows that the effect of collisions is negligible. In order to schedule a transaction, multi-site Gargamel has to synchronises schedulers to agree on the serial order of conflicting transaction. Each scheduler optimistically executes the transaction according to its local view, and in parallel, synchronises with the other schedulers. In most cases, by the time a transaction starts executing, the schedulers have already synchronised. Otherwise, if after transaction execution the agreement it not reached, the commitment is delayed until they do. In case of collision, a transaction should be canceled and rescheduled. Optimistic scheduling is especially suited for a geo-replicated setting, where the high communication latency between distant datacenters makes synchronisation slow. In this case, the optimistic approach mitigates the cross-datacenter synchronisation latency.

Since Gargamel runs at the load balancer, it does not impose any change to existing DBMS engines. Indeed, our prototype uses an unmodified PostgreSQL database.

Our single-site prototype shows that, under high incoming rates and with a good classifier, Gargamel performs considerably better than a Round-Robin certification-based system: it provides higher throughput, lower response time, better scalability, and consumes fewer resources.

We also discussed and measured the advantages of passive replication over active replication (known also as state machine replication) in the TPC-C workload.

We show that Gargamel is able to scale, parallelising update transactions while maintaining the consistency guarantees of Parallel Snapshot Isolation.

In multi-site settings we show by simulation the benefits of the optimistic concurrency control. We are actually working at the comparison between simulation and implementation results for multi-site gargamel.

8.2 Perspectives

Avoiding certification is one of the main benefits of our approach, one of the perspective of this thesis is to investigate how to circumvent or limit certification in case of false positives. An imperfect classifier may causes false positives, which serialises some transactions that could execute in parallel. This results in lower performance than the theoretical maximum. Our approach can be extended with machine-learning techniques, in order to learn from the classifier's past mistakes. However, this may also cause false negatives, which currently impose a certification step at the end of transaction execution.

An open perspective is to study a scheduling policy for multi-site Gargamel that favors performance over resource utilisation, executing the prefix of split chains at several sites in parallel. The current prototype optimises resource utilisation: a site that is in charge of the part of chain following a split point, waits to receive the write-set from the site executing the prefix before the split point. To improve responsiveness, we can execute the prefix in parallel at all nodes, such that if a node is less loaded or is faster

than the others, it will send its write-set early, allowing the slower node(s) to catch up. When the fastest node sends its write-set of the prefix, the others discard their ongoing execution and apply the received write-set.

Another research axis is to implement and compare other isolation levels. With a stronger isolation level, as 1-Copy SER, we expect Gargamel to have lower throughput, because of lower parallelism. In this case, we expect that the comparison with a certification-based system will be even more favorable to Gargamel, because of the high abort rate imposed by stricter isolation. Conversely, with an isolation level weaker than PSI, we expect Gargamel to increase its throughput by using more workers, but comparison with a certification-based system to be less favorable to Gargamel, because the former will suffer fewer aborts. We expect Gargamel to outperform certification-based systems, in terms of resource utilisation, until the isolation level is weak enough that certification does not aborts despite high parallelisation.

In general, we expect Gargamel to outperform a certification-based system in terms of throughput and latency until: (i) the certification-based system does not suffer aborts and (ii) it is faster to certify a transaction than to schedule it. The point (ii) depends on several factors, such as the degree of replication and the communication latency. Generally, certification performance degrades with the number of replicas and communication latency. In contrast, Gargamel scheduling algorithm degrades with the number of active transactions and with the inverse of conflict probability. paradoxically , the less a transaction is likely to conflict, the longer it takes to schedule it, because of the linear exploration of the graph. We plan to explore the certification/scheduling time trade-offs in future works.

A research direction is to investigate how Gargamel performs on top of a transactional key-value store as REDIS [16], Berkeley DB [44], Scalaris [55] or G-Store [21]. Generally, executing transactions in key-value stores is faster than in classical ACID databases. This changes the impact of aborts. The faster the transaction, the less wasteful is aborting, so the benefit from not aborting is smaller. Moreover, the fact that trans-

actions execute fast can make a state-machine replication approach attractive, rather than passive replication. The interest of state machine replication is to scale read-only transactions or use partial replication. Since state machine replication re-execute update transactions are at every replica, to add more replicas does not improve throughput, unless replication is partial. If so updates will affect only those replicas that store the updated records, allowing the system to scale the update workload. We plan to investigate how to support partial replication, and to deploy Gargamel on top of a partially-replicated key-value store. Support for partial state machine replication will allow Gargamel to be effective in workloads with fast transactions that does not benefit from the passive replication schema currently adopted by Gargamel.

Deuxième partie

Résumé de la thèse

Chapitre 9

Résumé de la Thèse

Contents

9.1 Introduction 126

9.1.1 Définition du Problème 127

9.1.2 Contributions : Gargamel 129

9.2 Gargamel Mono-Site 131

9.2.1 Architecture du Système 132

9.2.2 Modèle de système 133

9.2.3 Conclusion 133

9.3 Gargamel Multi-Site 134

9.3.1 Introduction 134

9.3.2 Architecture du Système 135

9.3.3 Conclusion 136

9.4 Simulation 137

9.5 Implementation de Gargamel 139

9.6 Conclusion 139

9.1 Introduction

Les bases de données sont utilisées depuis longtemps pour stocker et manipuler des données. Elles fournissent aux applications une manière simple et puissante pour traiter leur données. Leur succès est principalement dû au fait que les bases de données exposent une interface (typiquement SQL[1]) qui masque aux utilisateurs et aux applications l'emplacement et l'organisation interne des données. Grâce à la séparation de la définition logique des données et de leur gestion, les applications n'ont pas besoin de connaître ni où ni comment les données sont stockées. Les données peuvent être stockées sur disque, en mémoire, localement ou à distance, elles peuvent être répliquées ou non. Tout ceci se fait de manière transparente pour l'application. Les DBMSs classiques fournissent des garanties fortes et une sémantique transactionnelle complète qui facilite grandement les accès et les traitements des données pour les applications.

La nature omniprésente des bases de données rend leur tolérance aux pannes (i.e., la capacité à réagir avec grâce face à une panne) et leurs performances (en termes de débit et temps de réponse) critiques. La réplication est souvent utilisée à la fois pour faire face aux pannes et pour améliorer les performances. La réplication de bases de données a le potentiel d'améliorer les performances et la disponibilité, en permettant à plusieurs transactions de s'exécuter en parallèle sur des machines différentes.

La réplication de bases de données fonctionne bien pour les transactions en lecture seule mais les mises-à-jours restent problématiques. En effet, le contrôle de la concurrence est un mécanisme coûteux. Il est également inefficace d'exécuter des transactions conflictuelles de manière concurrente, parce que au moins une d'entre elles devra être abandonnée et recommencer. Ce problème bien connu empêche les bases de données d'être exploitées efficacement sur des architectures modernes comme les multi-cœurs, les clusters, les grilles et les clouds.

[1]SQL (pour l'anglais Structured Query Language) est un langage normalisé servant à exploiter des bases de données relationnelles.

Cette thèse s'intéresse au passage à l'échelle des bases de données répliquées sur un nombre potentiellement élevé de répliques avec un support efficace pour les écritures, sans pour autant renoncer à la cohérence.

9.1.1 Définition du Problème

Toutes les bases de données ont un système de contrôle de la concurrence pour assurer l'isolation des transactions. Le contrôle de la concurrence coordonne les transactions s'exécutant en parallèle afin d'éviter les anomalies et maintenir les invariants. Le niveaux d'isolation des transactions détermine le niveaux de cohérence fourni par une base de données.

Le critère traditionnel de justesse d'un mécanisme de contrôle de la concurrence est la *sérialisabilité*. Ce critère signifie que l'*historique d'exécution* (i.e., la séquence d'opérations de lecture/écriture effectuée) est équivalent à une certaine histoire séquentielle.

Dans les bases de données repartie la concurrence n'est pas seulement "au sein" des répliques, mais aussi "entre" les répliques de base de données ; le contrôle de la concurrence est donc beaucoup plus difficile parce que il doit coordonner l'exécution à la fois dans et entre les répliques. Un critère de justesse naturel est *1-copy-serialisability*. *1-copy-serialisability* signifie que l'exécution d'une base de données repartie est indistinguable de l'exécution d'une seule base de données avec serialisation comme niveaux d'isolation.

L'inconvénient de ce critère strict de justesse (aussi appelé "cohérence forte") est qu'il s'oppose à la capacité de passer à l'échelle. Un autre goulot d'étranglement que nous abordons dans cette thèse vient du fait qu'en utilisant un mécanisme de réplication totale (i.e., toutes les répliques contiennent toutes les données) une base de données repartie dois se synchroniser avec toutes les répliques à chaque fois qu'une transaction effectue une mise-à-jour.

Il existe plusieurs façon d'aborder le passage à l'échelle de bases de données ré-

parties. Le goulot d'étranglement du contrôle de la concurrence peut être atténué en relâchant le niveau d'isolation [25, 60, 12], en relâchant les proprietés transactionnelles ACID [63, 23, 1, 23], en parallélisant les lectures [48, 50, 58] ou encore en utilisant des mécanismes de réplication partielle [59, 56, 8, 3]. Stonebraker et al. dans [63] affirment que les DBMSs actuelles doivent tout simplement être retirées et remplacées par une collection de DBMSs spécialisées optimisées pour les différents champs d'application comme le texte, l'entrepôt de données et le processing de flux. Toutes ces approches viennent avec leurs avantages et inconvénients.

Les approches susmentionnées ne fonctionnent bien que pour certaines classes d'applications : relâcher le niveau d'isolation peut introduire des anomalies qui peuvent potentiellement violer des invariants applicatifs. Renoncer aux propriétés ACID transactionnelles rend le développement des application plus compliqué. La parallélisation des lectures est une bonne approche pour les applications produisant essentiellement des lectures mais ne permet pas de faire passer à l'échelle lorsque les charges ont beaucoup d'écritures. La réplication partielle est complexe à mettre en œuvre : les transactions faisant intervenir plusieurs partitions ne sont pas bien supportées et sont potentiellement inefficaces. Le coût de développement des bases de données spécialisées ne peut pas être reparti sur un grand nombre d'utilisateurs comme celui de bases de données banalisées, développer des bases de données spécialisées est donc souvent moins intéressant économiquement que développer des bases de données banalisées.

Notre proposition est de maintenir les garanties de cohérence traditionnelles fournies par les bases de données commerciales et de fournir de fortes garanties transactionnelles. Nous évitons la synchronisation des répliques suite à chaque mise-à-jour en déplaçant le système de contrôle de la concurrence *avant* l'exécution des transactions, au niveau du répartiteur de charge (*load balancer*).

9.1.2 Contributions : Gargamel

Les bases de données reparties ne passent souvent pas bien à l'échelle, à cause du coût du contrôle de la concurrence et de la contention de ressources.

Nous avons observé qu'il est plus efficace exécuter les transactions conflictuelles séquentiellement qu'en parallèle parce que l'exécution de transactions conflictuelles en parallèle provoque des abandons, et génère un gaspillage de travail.

Notre solution consiste en classifier les transactions selon les conflits prévus. Cette classification est faite dans le frontal de la base de données répliquée. Les transactions non-conflictuelles peuvent s'exécuter en parallèle dans des répliques différentes, assurant un haut débit ; toutes les transactions, celle en écriture comme celle en lecture, peuvent être parallélisées. Les transactions qui sont en conflits sont exécutées séquentiellement, assurant qu'elles ne seront pas abandonnées, optimisant ainsi l'utilisation des ressources. Cette stratégie est flexible et laisse les applications choisir le niveau d'isolation adéquat pour leurs besoins. Nous traitons le niveau d'isolation de notre prototype dans le Chapitre 6. Gargamel est conçu pour permettre au système d'exécuter les transactions de façon asynchrone ; il n'exige pas une (coûteuse et lente) synchronisation globale. Notre système fonctionne comme un frontal à une base de données non-modifiée, évitant le coût d'utilisation de verrous, conflits, et abandons. Notre système améliore également la localité : en effet Gargamel partitionne la base de données dynamiquement selon les types de transactions. Tout cela donne lieu à un meilleur débit, un meilleur temps de réponse et une meilleure utilisation des ressources : nos expérimentations montrent une amélioration considérable pour les charges présentant une forte contention et une perte négligeable dans les autres cas.

Notre classificateur actuel est basé sur une analyse statique du texte des transactions (procédures enregistrées ou *stored procedures*). Cela est réaliste car la logique de nombreuses applications (e.g., sites pour le commerce électronique) est comprise dans un petit ensemble fixe de transactions parametrisées, et l'accès ad-hoc à la base de données est rare [63]. Notre analyse statique du benchmarck TPC-C est complète, i.e., il n'y a

pas des faux négatifs : si un conflit existe, il seras prédit. Cependant, des faux positifs sont possibles : un conflit peut être prédit même s'il ne s'avère pas à l'exécution. S'il y a des faux positifs, Gargamel sérialise plus que nécessaire parce que il peut sérialiser des transactions qui ne sont pas effectivement en conflit. Pour des autres types de cas d'usages, si des faux négatifs ne peuvent pas être évités, Gargamel peut tout de même être utilisé. Cependant, il doit certifier les transactions après leur exécution, parce que des transactions conflictuelles peuvent alors être exécutées de façon concurrente.

Nos contributions sont les suivantes :

- Nous montrons comment paralléliser les transactions non conflictuelles en ajoutant à la base de données un classificateur de transactions, et nous détaillons l'algorithme d'ordonnancement correspondant. Toutes les répliques de la base de données s'exécutent séquentiellement sans contention de ressources.

- Nous proposons un prototype de classificateur simple, basé sur une analyse statique du texte de procédures enregistrées.

- Nous avons implémenté un simulateur à évènements discrets qui a été utilisé pour valider rapidement les premières idées. Les résultats de simulations sont publiés dans [20].

- Nous avons ensuite démontré l'efficacité de notre idée avec un prototype, en variant plusieurs paramètres et en se comparant avec un ordonnanceur Round-Robin.

- Nous concluons de l'évaluation que (i) quand la charge est importante, Gargamel améliore la latence d'un ordre de grandeur, dans le benchmark TPC-C par rapport a un système Round-Robin. Quand la charge est moins importante, Gargamel n'apporte pas d'amélioration, mais le sur-coût induit est négligeable. (ii) Gargamel nécessite moins de ressources, diminuant leur coût, et passe à l'échelle beaucoup mieux que le système Round-Robin.

9.2 Gargamel Mono-Site

Coordonner l'accès concurrent à une base de données repartie est difficile. Une partie de la complexité viens du mécanisme de contrôle de la concurrence. Il existe plusieurs techniques (e.g., utilisation de verrous et/ou de communications de groupe) pour garder la cohérence des répliques. Leur complexité est corrélée au nombre de répliques.

Dans la vie d'une transaction it est possible de distinguer une phase d'exécution et une phase de certification. La phase d'exécution est le temps utilisé par la base de données pour exécuter le code de la transaction. La phase de certification est le temps usé par le mécanisme de contrôle de la concurrence pour vérifier si la transaction doit être validée (commit) ou être abandonnée.

La phase de certification est essentielle pour la justesse. S'il y a une transaction de mise-à-jour, elle ne peut pas être évitée. Généralement elle est exécutée après l'exécution de la transaction pour décider si le système doit valider ou abandonner la transaction (même si des systèmes peuvent abandonner des transactions préventivement pendant la phase d'exécution). Ce mécanisme amène un problème de passage à l'échelle : il doit être coordonné entre toutes les répliques en cas de réplication totale et entre toute les répliques qui stockent les données accédées en cas de réplication partielle. Cela provoque un goulot d'étranglement dans la synchronisation, qui s'aggrave lorsque le nombre de répliques ou la latence inter-répliques augmente.

Notre observation clef est que, pour être optimal en termes de débit et d'utilisation de ressources, nous avons besoin de :

I. ne jamais exécuter concurremment des transactions conflictuelles parce qu'elle serons abandonnées, générant un gaspillage des ressources utilisées pour exécuter (optimalité de ressources) ;

II. exécuter en parallèle les transactions qui ne sont pas en conflit (optimalité de débit)

Pour accomplir ces objectifs, nous utilisons un mécanisme d'estimation des conflits,

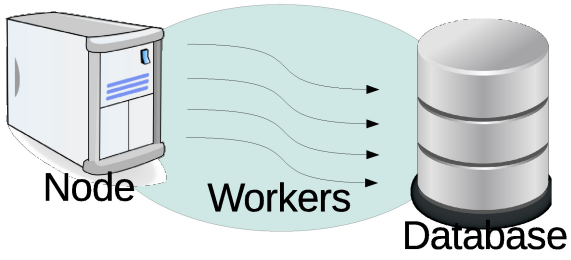


FIGURE 9.1 : Node and Workers architecture

pour faire le contrôle de la concurrence de manière pessimiste, en amont de l'exécution des transactions. Grâce à l'estimation des conflits, nous pouvons exécuter les transactions conflictuelles séquentiellement dans la même réplique, et distribuer les transactions non conflictuelles sur plusieurs répliques pour les exécuter en parallèle [20]. Si l'estimation est complète (i.e., il n'y a pas de faux négatifs ou faux positifs) alors l'exécution sera optimale en terme de débit et d'utilisation de ressources. En outre, si il n'y a pas de faux négatifs, Gargamel n'a pas besoin de certifier les transactions après leur exécution : elle ne seront pas en conflit avec une transaction concurrente. Une fois la transactions exécutée, son ensemble d'écriture (write-set) est propagé à toutes les autres répliques de manière fiable et en ordre total.

9.2.1 Architecture du Système

Classiquement, les architecture de bases de données reparties sont constituées de clients, qui envoient les requêtes à un serveur frontal, que les re-envoient à un ensembles de répliques (nœuds). Tous les nœuds contiennent une réplique de l'entière base de données. Gargamel est basé sur cette architecture. Il est localisé dans le serveur frontal et n'exige aucune modification à la base de données sous-jacente.

9.2.2 Modèle de système

Nous appelons le frontal *Ordonnanceur Gargamel* (ou simplement ordonnanceur) et nous appelons *Nœuds* les répliques qui stockent la base de données. Tous les nœuds ont un ou plusieurs processus qui accèdent à la base de données (en général un par cœur). Nous appelons ces processus *Workers* (comme illustré par la Figure 9.1). La base de données est entièrement répliquée dans chaque nœud.

Les ordonnanceurs, les nœuds et les clients communiquent par passage de messages. Le système de communication de group utilisé offre un ordre de livraison FIFO (premier entré, premier sorti) et ABCAST (ordre total avec livraison atomique). Les communications entre l'ordonnanceur et les nœuds, l' ordonnanceur et les clients, et entre les nœuds et les clients sont faites avec des canaux de communications FIFO. Les communications entre les nœuds utilisent des canaux de communications ABCAST pour garder un ordre total dans la propagation des write-set des transactions.

Le composant qui prédis les conflits éventuels entre transactions est appelé *classificateur de transactions*. Notre classificateur de transactions courant est basé sur une analyse statique du texte des transactions (procédures enregistrées). Cette stratégie est réaliste pour les applications qui utilisent les procédures enregistrées pour accéder à la base de données.

9.2.3 Conclusion

Gargamel est conçu pour assurer le passage à l'échelle des bases de données reparties utilisant un mécanisme de réplication totale. Notre étude expérimentale confirme les bénéfices offerts (voir le chapitre 7). Le but est obtenu grâce à un contrôle de la concurrence pessimiste qui agit à priori pour éliminer les abandons des transactions. Si le classificateur ne souffre pas de faux négatifs, il élimine la nécessité de la phase de certification après l'exécution des transactions. L'avantage en termes de performances et utilisation des ressources dépend de la précision du classificateur et de caractéristiques des ac-

cès. Gargamel offre la flexibilité pour implémenter plusieurs niveaux d'isolation pour répondre aux besoins des applications avec une synchronisation minime.

Puisque Gargamel fais le contrôle de la concurrence à l'avance, avant l'exécution des transactions, il a besoin de faire une estimation des écritures et lectures au moment ou la transaction est soumise au système. Les transactions interactives (i.e., les clients envoient une séquence de commandes de lecture et écritures capsulées entre un commande se start- et un de commit-transaction) ne sont pas supportées. À notre avis, cette restriction est rentable et n'est pas trop restrictive pour les applications de type OLTP.

Notre classificateur de transactions "proof of concept" utilise une simple analyse statique du workload. Il contrôle les paramètres des stored procedure pour prédire les conflits. Nous montrons un classificateur pour le benchmark TPC-C qui est "sound", i.e., s'il y a un conflit il serais prévu, toutefois, il peut avoir des faux positifs (il peut prédire un conflit qui ne se produira pas pendant l'exécution). Nous avons discuté les effets d'un classificateur imparfait : les faux positifs impliquent que certaines transactions soient sérialisée même si elle pourraient être exécutées en parallèle. Ceci conduit à des performances au-dessous du maximum théorique. Nous envisageons d'étendre notre technique courante avec des techniques d'apprentissage. Par contre, cette technique peut amener à des faux négatifs, qui imposent de certifier les transactions après l'exécution pour éviter de valider des transactions concurrentes conflictuelles.

9.3 Gargamel Multi-Site

9.3.1 Introduction

Dans la Section 9.2 nous avons décrit une architecture avec une base de données répliquée dans an seul site (i.e., centre de calcul) focalisant la discussion sur la capacité de passer à l'échelle et sur une utilisation des ressources optimale. Ces objectifs sont atteints grâce à un contrôle de la concurrence pessimiste. Le contrôle de la concurrence

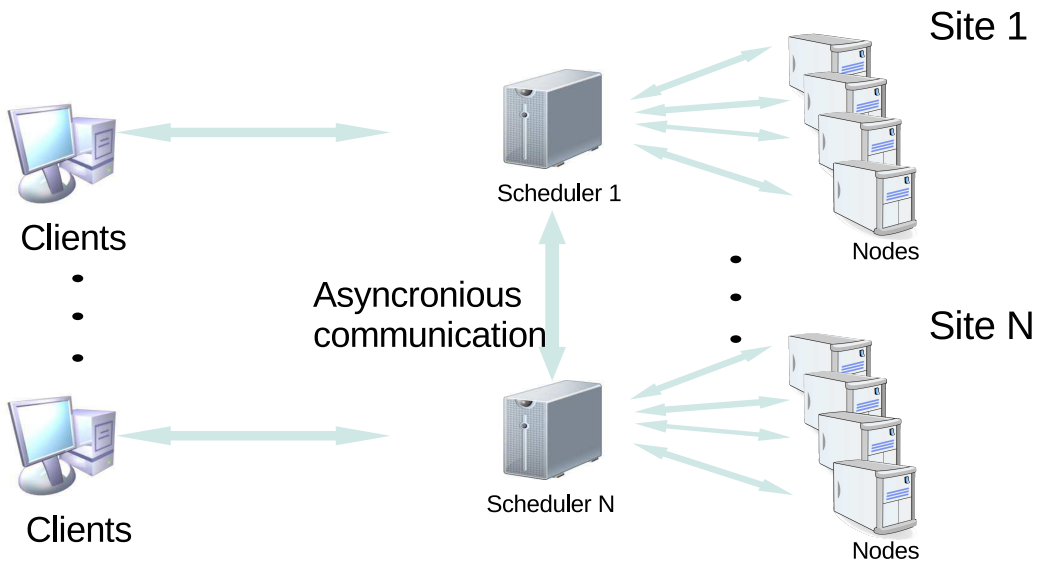


FIGURE 9.2 : Multi-site system architecture

pessimiste sérialise les transactions conflictuelles pour éviter les abandons et réparti les transactions pas conflictuelles.

Ici, nous traitons une architecture multi-site, avec un ordonnanceur par site. Cette approche à le potentiel de diminuer la latence du client qui peut se connecter à un site plus proche. Cette architecture demande de synchroniser les ordonnanceurs pour éviter qu'ils ne divergent. Pour éviter de pénaliser le débit du système, les ordonnanceurs se synchronisent de manière optimiste, hors du chemin critique.

9.3.2 Architecture du Système

Gargamel multi-site, comme illustré par la Figure 9.2, est composé de plusieurs sites, avec chacun un ordonnanceur et un ensemble de nœuds.

Un site peut être, par exemple, un puissant multicore dans un centre de calcul ou un centre de calcul dans un cloud. L'importante étant que le temps d'envoi d'un message

envoyé par un site à un autre, la *latence inter-site*, soit beaucoup plus importante que la latence entre un ordonnanceur et un worker local. Nous rappelons que un worker est un processus qui accéde la base de données. Dans notre expérimentation un nœud a un worker par CPU.

Tous les ordonnanceurs gèrent l'exécution des transactions dans leur nœuds comme décrit dans la Section 9.2 : à l'intérieur d'un site, les transactions conflictuelles sont sérialisées et celle non conflictuelles sont réparties entre les répliques.

Comme dans le cas mono-site, un ordonnanceur reçoit les transactions depuis ses clients locaux et les envoie à ses nœuds pour les exécuter en parallèle. De plus, un ordonnanceur dois synchroniser sa vue locale de conflits des transactions avec les autres ordonnanceurs pour inclure les conflits avec les transactions ordonnées dans les ordonnanceurs distants. La synchronisation entre ordonnanceurs est optimiste i.e., un ordonnanceur envoie d'abord la transaction au worker, puis se synchronise avec les autres ordonnanceurs.

Nous envisageons plusieurs cas d'usages où une configuration multi-site peut s'avérer utile. Par exemple, si la latence entre le client et l'ordonnanceur est haute, il peut être avantageux de créer un site proche du client. Cela va permettre de baisser la latence perçue par le client. Un autre cas est quand la charge dépasse la capacité d'un seul site ; ou quand il est nécessaire d'offrir plus de disponibilité et les répliques doivent être répandue dans plusieurs lieux distants.

9.3.3 Conclusion

Gargamel multi-site permet à plusieurs sites geo-répliquées, chacun composé d'un ordonnanceur et d'un ensemble de nœuds, de s'exécuter en parallèle. Chaque site reçoit les transactions depuis les clients locaux et les exécute dans les nœuds locaux. La synchronisation entre les sites sur l'ordre d'exécution est optimiste, hors du chemin critique.

Gargamel multi-site est approprié pour baisser la latence perçue par les clients en plaçant les ordonnanceurs proche d'eux, pour améliorer la disponibilité en plaçant les ordonnanceurs dans de régions géographiques multiples et pour étendre le système quand la charge dépasse la capacité d'un seul site.

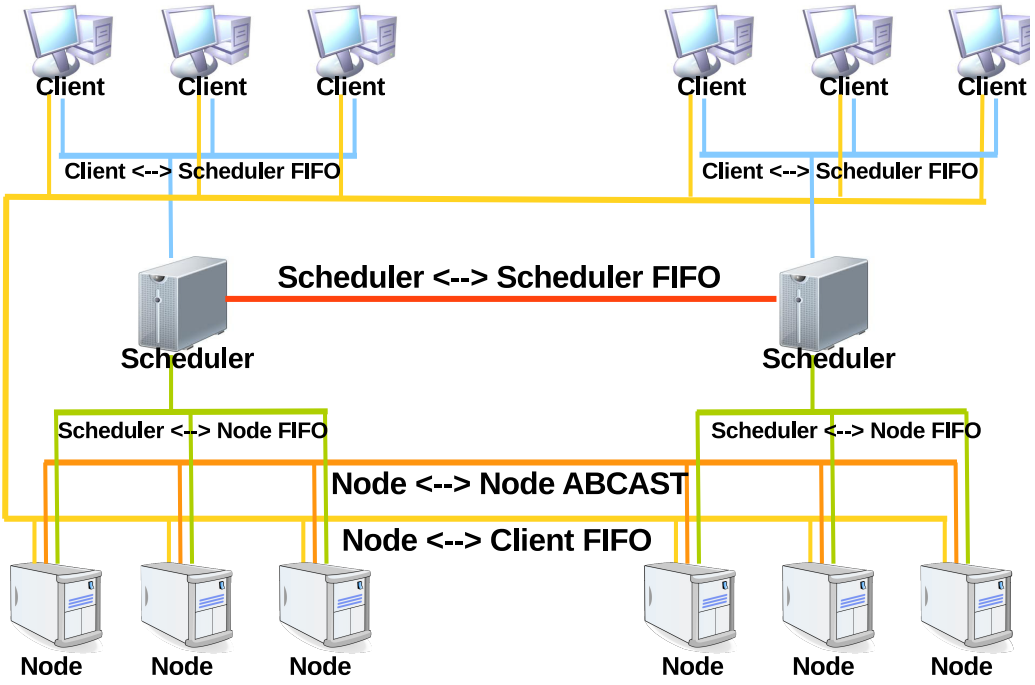
Nous avons évalué les performances et les bénéficies de Gargamel avec un simulateur à évènements discrets. Les résultats de simulations sont présentés dans la prochaine section.

9.4 Simulation

Nous avons implémenté un simulateur à évènements discrets pour analyser les avantages et les limites de l'architecture de Gargamel avant de continuer avec un mise en œurve.

L'objectif de la simulation est d'évaluer (i) le temps de réponse et le débit des transactions de mise-à-jour, (ii) le sur-coût induit, (iii) le besoin de ressources, (iv) l'impact des conflits dans le cas multi-site, (v) la précision du classificateur.

Le simulateur a été utile pour expérimenter rapidement différentes stratégies de réplication (comme primary-copy ou update-anywhere), différents contrôle la concurrence et niveaux d'isolation. Nous avons entre-autre testé la faisabilité d'un classificateur pour deux benchmarks différents. Les résultats de simulation nous ont motivé pour implémenter un prototype complet. Plusieurs choix de design ont été basés sur les résultats de simulation. C'est le cas, par exemple, de l'utilité d'une stratégie "update-anywhere" et le choix de synchroniser les ordonnanceurs de façon optimiste dans le cas multi-site.



Canaux de communication entre les nœuds, les clients et l'ordonnaceur.

client <–> scheduler Canal FIFO utilisé par les clients pour envoyer les requêtes d'exécution des transactions à l'ordonnanceur. L'ordonnanceur utilise ce canal seulement une fois pour envoyer un message "starts" au client.

scheduler <–> node Canal FIFO utilisé par l'ordonnanceur pour re-envoyer les requêtes d'exécution des transactions au nœud choisi. Le message de requête d'exécution de la transaction contient aussi les dépendances de la transaction.

node <–> node Canal ABCAST utilisé par les nœuds pour diffuser le write-set des transactions exécutées et pour envoyer le message de certification.

node <–> client Canal FIFO utilisé par les nœuds pour répondre aux clients quand une transaction est validée. Les clients ne communiquent jamais avec les nœuds en utilisant ce canal.

scheduler <–> scheduler FIFO Canal FIFO utilisé par les ordonnanceurs pour synchroniser le graphe des transactions.

FIGURE 9.3 : Communication channels

9.5 Implementation de Gargamel

Pour évaluer Gargamel nous avons construit un prototype qui fonctionne en utilisant une base de données non modifiée. L'objectif de l'implementation est d'évaluer le système dans un environnement réel plutôt que de valider le simulateur.

Les principaux composants du prototype (ordonnanceur, nœuds et clients) sont écris en Java (∼12k lignes de code). Ils communiquent à travers JGroups [42], un système de multicast fiable utilisé par JBoss [27], un serveur d'applications Java open-source (maintenant renommé en WildFly). Le contrôle de la concurrence et le mécanisme de propagation des mises-à-jours sont basés sur la communication de groupes (group communication), et la justesse est basée sur les propriétés des canaux de communication. La Figure 9.3 donne une vue d'ensemble des canaux de communication. Il y a un canal ABCAST et trois canaux FIFO. Le canal ABCAST relie les nœuds entre eux, un canal FIFO relie les ordonnanceur et les nœuds, un canal FIFO relie l'ordonnanceur avec les clients, un autre canal FIFO relie les nœuds et les clients, et un dernière canal FIFO relie les ordonnanceurs entre eux.

9.6 Conclusion

Notre système, Gargamel, est un intergiciel que maximise le parallélisme dans les base de données réparties tout en évitant de gâcher du travail. À la place de paralléliser toutes les transactions, y compris celle conflictuelles, comme la plus part des systèmes précédents, Gragamel pre-sérialise les transactions conflictuelles. L'idée clé est de vérifier les conflits avant l'exécution des transactions et de les ordonner selon un graphe de dépendance des transactions. Les transactions sont ordonnées de telle manière que les transactions conflictuelles ne sont jamais exécutées concurremment, et que les transactions non conflictuelles sont parallélisées. L'avantage de ce choix est triple :

Maximiser le parallélisme : Les transactions non conflictuelles sont réparties sur des

répliques différentes pour être exécutée en parallèle.

Eviter de gâcher le travail : Les transactions ne sont jamais abandonnées et re-exécutées.

Minimiser la synchronisation : L'exécution des transactions est libre de conflits, donc il n'y a pas besoin de certification.

Ces propriétés améliorent le temps de réponse, le débit, et le passage à l'échelle, tout en diminuant l'utilisation des ressources par rapport à un système qui se base sur la certification des transactions.

La précision du classificateur impacte la quantité de parallélisme et de travail gaspillé. Si le classificateur souffre des faux positifs, le parallélisme peut être sub-optimal parce que des transactions non conflictuelles peuvent être inutilement sérialisées. Si il souffre de faux négatifs, l'utilisation de ressources est sub-optimale parce que des transactions peuvent être abandonnées. En outre, avec des faux négatifs, Gargamel ne peut pas éviter la certification, qui est coûteuse.

Notre classificateur "proof-of-concept" utilise une analyse statique simple du des accès. Nous proposons un classificateur pour le benchmark TPC-C, et nous discutons comment éviter les faux négatifs et réduire les faux positifs. Nous discutons aussi les effets d'un classificateur imparfait.

Dans le cadre du multi-site, Gargamel gère plusieurs ordonnanceurs qui s'exécutent de manière optimiste, en parallèle, pouvant donner lieux à des collisions. Nous montrons avec notre simulateur que l'effet des collisions reste négligeable. Pour ordonner une transaction, Gargamel multi-site doit synchroniser les ordonnanceurs afin de s'accorder sur l'ordre des transactions conflictuelles. Chaque ordonnanceur exécute les transactions de manière optimiste selon sa vue locale, et en parallèle, il se synchronise avec les autres ordonnanceurs.

Dans la plus part des cas, la synchronisation s'effectue le temps que la transaction s'exécute. Sinon, si après l'exécution d'une transaction Gargamel n'est pas arrivé à un

accord, la validation est remise jusqu'à ce que l'accord soit atteint. En cas de collision une transaction est annulée et re-ordonnée. L'ordonnancement optimiste est notamment adapté dans le cadre de la géo-réplication, ou la grosse latence de communication entre centres de calcul distants rend la synchronisation lente. Dans ce cas, la stratégie optimiste atténue la latence de synchronisation à travers les centres de calcul.

Puisque Gargamel s'exécute comme en frontal, il ne nécessite aucune modification des bases de données existantes. D'ailleurs, notre prototype utilise comme base de données une version non modifiée de PostgreSQL.

Notre prototype en mono-site montre que, avec une forte charge et avec un bon classificateur, Gargamel fonctionne offre de bien meilleures performances qu'un système Round-Robin basé sur la certification : Gargamel a un plus haut débit, un meilleur temps de réponse, une meilleure capacité à passer à l'échelle et utilise moins de ressources.

Nous avons aussi mesuré les avantages de la réplication passive par rapport à la réplication active (connue aussi comme *state machine replication*) dans le workload de TPC-C.

Nous avons montré que Gargamel est capable de passer à l'échelle, en parallélisant les transactions de mises-à-jours, tout en gardant les garanties de cohérence de Parallel Snapshot Isolation.

Dans le cas du multi-site nous avons montré avec la simulation les bénéfices du contrôle de la concurrence optimiste.

Bibliography

Bibliography

[1] Memcachedb http://memcachedb.org/.

[2] SQL server 20014 transaction isolation levels. Microsoft Developer Network.

[3] Gustavo Alonso. Partial database replication and group communication primitives (extended abstract). In *in Proceedings of the 2 nd European Research Seminar on Advances in Distributed Systems (ERSADS'97*, pages 171–176, 1997.

[4] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[5] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In Markus Endler and Douglas C. Schmidt, editors, *Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 282–304. Springer, 2003.

[6] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010.

[7] Methods to implement the random database population. http://db.apache.org/derby/javadoc/testing/org/apache/derbyTesting/system/oe/util/OERandom.html.

[8] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. González de Mendívil, and F. D. Muñoz Escoí. Sipre: A partial database replication protocol with si replicas. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 2181–2185, New York, NY, USA, 2008. ACM.

[9] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: virtues and limitations. In *40th International Conference on Very Large Data Bases (VLDB)*, 2014.

[10] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with ramp transactions. In *SIGMOD Conference*, pages 27–38, 2014.

[11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. pages 229–240, Asilomar, CA, USA, January 2011.

[12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24:1–10, May 1995.

[13] Philip A. Bernstein and Nathan Goodman. Serializability theory for replicated databases. *Journal of Computer and System Sciences*, 31(3):355 – 374, 1985.

[14] Navin Budhiraja, Keith Marzullo Y, Fred B. Schneider Z, and Sam Toueg X. Chapter 8: The primary backup approach, 1999.

[15] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[16] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[17] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[19] Bernadette Charron-Bost, Fernando Pedone, and Andre Schiper. *Replication: theory and Practice*, volume 5959. springer, 2010.

[20] Pierpaolo Cincilla, Sébastien Monnet, and Marc Shapiro. Gargamel: boosting DBMS performance by parallelising write transactions. In *Parallel and Dist. Sys. (ICPADS)*, pages 572–579, Singapore, December 2012.

[21] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *SoCC*, pages 163–174. ACM, 2010.

[22] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 715–726. VLDB Endowment, 2006.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[24] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: memory-aware load balancing and update filtering in replicated databases. *SIGOPS Oper. Syst. Rev.*, 41(3):399–412, June 2007.

[25] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. *Reliable Distributed Systems, IEEE Symposium on*, 0:73–84, 2005.

[26] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[27] Marc Fleury and Francisco Reverbel. The JBoss extensible server. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 344–373, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[28] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[29] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25:173–182, June 1996.

[30] The PostgreSQL Global Development Group. PostgreSQL 9.3.4 documentation, September 2013.

[31] R. Hecht and S. Jablonski. NoSQL evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341, Dec 2011.

[32] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[33] Hewlett-Packard. TPC benchmark C: full disclosure report for HP ProLiant BL685c G7, 2011.

[34] IBM. TPC benchmark E: Full disclosure report for IBM System x 3850 X5, 2011.

[35] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc., http://aws.amazon.com/ec2/#pricing, 2008.

[36] K. Jacobs. Concurrency control: Transaction isolation and serializability in SQL92 and Oracle7. Oracle White Paper, July 1995. Part No. A33745.

[37] Bettina Kemme and Gustavo 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.

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

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

[40] Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fabio Porto, and Patrick Valduriez. Dynamic workload-based partitioning algorithms for continuously growing databases. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XII*, volume 8320 of *Lecture Notes in Computer Science*, pages 105–128. Springer Berlin Heidelberg, 2013.

[41] Diego R. Llanos. TPCC-UVA: An open-source TPC-C implementation for global performance measurement of computer systems. *ACM SIGMOD Record*, December 2006. ISSN 0163-5808.

[42] Alberto Montresor, Renzo Davoli, and Özalp Babaoğlu. Middleware for dependable network services in partitionable distributed systems. *SIGOPS Oper. Syst. Rev.*, 35(1):73–96, January 2001.

[43] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[44] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. USENIX, 1999.

[45] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[46] Esther Pacitti, M. Tamer Özsu, and Cédric Coulon. Preventive multi-master replication in a cluster of autonomous databases. In *Euro-Par'03*, pages 318–327, 2003.

[47] Fernando Pedone, Rachid Guerraoui, and André Schiper. The Database State Machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[48] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *In Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, pages 155–174, 2004.

[49] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[50] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 17–30, New York, NY, USA, 2011. ACM.

[51] Idrissa Sarr, Hubert Naacke, and Stéphane Gançarski. DTR: Distributed transaction routing in a large scale network. In *High Performance Computing for Computational Science - VECPAR 2008*, volume 5336 of *Lecture Notes in Computer Science*, pages 521–531. Springer Berlin Heidelberg, 2008.

[52] André Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Commun. ACM*, 39(4):84–87, April 1996.

[53] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, May 1984.

[54] Fred B. Schneider. Replication management using the State Machine approach, 1993.

[55] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA, 2008. ACM.

[56] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, PRDC '07, pages 290–297, Washington, DC, USA, 2007. IEEE Computer Society.

[57] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 290–297, Dec 2007.

[58] Jason Sobel. Scaling out. Engineering @ Facebook Notes https://www.facebook.com/note.php?note_id=23844338919, August 2008.

[59] António Sousa, Rui Oliveira, Francisco Moura, and Fernando Pedone. Partial replication in the database state machine. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, NCA '01, pages 298–309, Washington, DC, USA, 2001. IEEE Computer Society.

[60] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[61] Michael Stonebraker. SQL databases v. NoSQL databases. *Commun. ACM*, 53(4):10–11, April 2010.

[62] Michael Stonebraker and Ugur Cetintemel. "One size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[63] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat 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*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[64] Michael Stonebraker and Lawrence A Rowe. The design of POSTGRES. In *ACM Sigmod Record*, volume 15, pages 340–355. ACM, 1986.

[65] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149, Sep 1994.

[66] TPC. TPC benchmark C Standard Specification Revision 5.11, 2001.

[67] TPC. TPC benchmark E Standard Specification Revision 1.12.0, 2010.

[68] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[69] Zhou Wei, G. Pierre, and Chi-Hung Chi. Cloudtps: Scalable transactions for web applications in the cloud. *Services Computing, IEEE Transactions on*, 5(4):525–539, Fourth 2012.

Glossary

ABCAST Atomic Broadcast (ABCAST) is a broadcast messaging protocol that ensures that messages are received reliably and in the same order by all participants.. 20

Back dependency Back dependency of a transactions t are all the transactions that are scheduled before t and conflict with t plural. 31, 33, 36, 37, 39, 40, 49, 55, 57

EC2 Amazon Elastic Compute Cloud (EC2) is a web service that offers resizable cloud hosting services. 68

FIFO First In, First Out (FIFO) is a broadcast messaging protocol that ensures that messages are received at the destination reliably and in the same order in which they are sent at the origin. 20

Front dependency Transactions with a write-set intersection with a transaction t that follow t in the scheduling plural. 31, 36, 37, 49

GTSN Global Transaction Sequence Number (GTSN) is an incremental global commit number that Gargamel associates to each committed transaction when certification is used. 93

Home transaction Transactions that schedulers receive from their clients. 48, 54, 57

Remote transaction Transactions that schedulers receive from other schedulers. 11, 33, 48, 49, 57

Round-Robin Round-robin algorithm assign transactions to each worker in equal portions and in circular order. iv, 7, 63, 68, 69, 72, 73, 76, 78, 91, 93, 97, 99, 100, 102, 104, 106, 107, 119, 130, 141

Tashkent+ memory-aware load balancing algorithm that minimize disk I/O. iv, 63, 68, 69, 72, 73, 76, 78

TPC-C On-line transaction processing benchmark. iv, 6, 7, 17, 35, 41, 63–67, 69–72, 74, 75, 77–85, 91–93, 95, 96, 98, 99, 101, 104, 110, 118, 119, 129, 130, 134, 140, 141

TPC-E On-line transaction processing benchmark. iv, 63–65, 67, 69, 81–88

Write-set Set of records written or updated by a transaction. 11, 16, 17, 20, 24, 29, 30, 32, 33, 37–40, 48, 54, 55, 58, 65, 90, 93–95, 104, 115, 119, 120, 132, 133, 138, 151

Acronyms

1-Copy SER 1-Copy Serialisability. 24, 37, 39, 120

ABCAST Atomic Broadcast. 20, 30, 40, 90, 91, 94, 95, 133, 138, 139

D Delivery. 64, 66, 67, 92, 101

DB database. 3–6, 9–13, 19–21, 23, 24, 28–33, 37–41, 46–48, 54, 56, 68, 69, 89, 91–93, 97, 101, 104, 110, 112–114, 118, 120

DBMS Database Management System. 3, 5, 16, 20, 21, 25, 91, 126, 128

EC2 Amazon Elastic Compute Cloud. 68, 76, 98, 107–109

FIFO First In First Out. 20, 30, 56, 90–92, 133, 138, 139

GSI Generalized Snapshot Isolation. 14, 15

GTSN Global Transaction Sequence Number. 93–95

JDBC Java Database Connectivity. 91, 93

LAN Local Area Network. 52, 78

MF Market Feed. 67

NO New Order. 64, 66, 67, 92, 102

NoSQL Not Only SQL. 21

OLTP online transaction processing. 23

OS Order Status. 64, 101

P Payment. 64, 66, 67, 92, 101

PCSI Prefix-Consistent Snapshot Isolation. 14, 15

PSI Parallel Snapshot Isolation. 15, 37–41, 96, 119, 120, 141

SI Snapshot Isolation. 13–15, 20, 32, 37–40, 64, 66, 67, 95, 96

SL Stock Level. 64, 101

TR Trade Result. 67

TU Trade Update. 67

WAN Wide Area Network. 24, 52, 78, 115

Pierpaolo CINCILLA

Gargamel: boosting DBMS performance by parallelising write transactions

Abstract

Databases often scale poorly in distributed configurations, due to the cost of concurrency control and to resource contention. The alternative of centralizing writes works well only for read-intensive workloads, whereas weakening transactional properties is problematic for application developers. Our solution spreads non-conflicting update transactions to different replicas, but still provides strong transactional guarantees. In effect, Gargamel partitions the database dynamically according to the update workload. Each database replica runs sequentially, at full bandwidth; mutual synchronisation between replicas remains minimal. Our prototype show that Gargamel improves both response time and load by an order of magnitude when contention is high (highly loaded system with bounded resources), and that otherwise slow-down is negligible.

Résumé

Les bases de données présentent des problèmes de passage à l'échelle. Ceci est principalement dû à la compétition pour les ressources et au coût du contrôle de la concurrence. Une alternative consiste à centraliser les écritures afin d'éviter les conflits. Cependant, cette solution ne présente des performances satisfaisantes que pour les applications effectuant majoritairement des lectures. Une autre solution est d'affaiblir les propriétés transactionnelles mais cela complexifie le travail des développeurs d'applications. Notre solution, Gargamel, répartie les transactions effectuant des écritures sur différentes répliques de la base de données tout en gardant de fortes propriétés transactionnelles. Toutes les répliques de la base de donnée s'exécutent séquentiellement, à plein débit; la synchronisation entre les répliques reste minime. Les évaluations effectuées avec notre prototype montrent que Gargamel permet d'améliorer le temps de réponse et la charge d'un ordre de grandeur quand la compétition est forte (systèmes très chargés avec ressources limitées) et que dans les autres cas le ralentissement est négligeable.