DOI: 10.15439/2021F34
ISSN 2300-5963 ACSIS, Vol. 25

Thespis: Causally-consistent OLTP

Carl Camilleri, Joseph G. Vella, Vitezslav Nezval
Computer Information Systems
University of Malta, Malta
Email: {carl.camilleri.04, joseph.g.vella, vitezslav.nezval}@um.edu.mt

***Abstract*—Data Consistency defines the validity of a data set according to some set of rules, and different levels of data consistency have been proposed. Causal consistency is the strongest type of consistency possible when data is stored in multiple locations, and fault tolerance is desired. Thespis is a middleware that leverages the Actor model to implement causal consistency over a DBMS, whilst abstracting complexities for application developers behind a REST interface. ThespisTRX is an extension that provides read-only transaction capabilities, whilst ThespisDIIP is another extension that handles distributed integrity invariant preservation. Here, we analyse standard transactional workloads on the relational data model, which is richer than the key-value data model supported by the Thespis interface. We show the applicability of the Thespis approach for this data model by designing new operations for the Thespis interface, which ensure correct execution of such workloads in a convergent, causally-consistent distributed environment.**

I. Introduction

THE CAP theorem [1], [2] proves that having both availability and partition tolerance within dispartite databases (DBs) that implement Strong Consistency (SC) [3] is not possible. SC is the strongest type of consistency offered by traditional distributed database management systems (DDBMSs), that manage databases across multiple nodes.

Distributed data centres have led to a wide adoption of DBs that forego strong data consistency in favour of availability and partition tolerance to provide the scalability and high availability properties sought by enterprise-scale applications. Popular DBs in this area offer Eventual Consistency (EC) [4], a weak consistency model which guarantees that given no new WRITE operations, all nodes (i.e. distributed partitions) of the DB eventually converge to the same state.

EC is relatively easy to achieve, and does not suffer from the performance limitations of distributed algorithms, such as Paxos [5], that attempt to achieve a degree of availability and SC in a distributed environment. However, EC shifts data safety and consistency responsibilities to the application layer, giving rise to a new set of problems [6].

Causal Consistency (CC) [7] is weaker than SC, but stronger than EC, and has been proven to be the strongest type of consistency that can be achieved in a fault-tolerant, distributed system [8]. Informally, CC implies that readers cannot find a version of a data element before all the operations that led to that version are visible [9].

II. Problem Domain

CC is sufficiently strong, and sufficiently performant, for most enterprise applications [10]. However, we believe its adoption in the industry is compounded by a number of aspects, including:

1) Lack of support for rich data modeling, with CC DBs supporting data sets based on the key-value data model, or abstract data types.
2) Programmer accessibility, given that existing CC DBs require engineering applications specifically around their semantics or client libraries, and do not sufficiently abstract the programmer from the complexities of distributed data management [11].
3) Database lock-in, with the CC DB storing data in native formats which are incompatible with other consumers.

We propose a middleware that achieves CC, stores the data in a relational database management system (RDBMS), and integrates with applications through intuitive APIs, abstracting the complexities of CC as much as possible. In this paper, we also analyse online transactional processing (OLTP) workloads that are traditionally used to benchmark widely-adopted RDBMSs, and propose the semantics that such a middleware should offer to enable an application to perform the same function within a CC DDBMS.

III. Definitions

This section identifies some terminology and provides relevant definitions, as used throughout the rest of the paper.

Replica. A replica denotes a copy of a database. Our context assumes that each replica is a full copy of the database.

Data Centre/Node. A Data Centre (DC), or node, refers to a physical location that hosts one of the replicas of a database.

Distributed Database. A distributed database (DDB) is considered to be a database which resides in multiple nodes.

Database Operation. A database operation denotes an activity that is performed by an application against some API offered by the database, or by the overlying middleware.

Operation Latency. The operation latency is the time elapsed between when a client submits a database operation to when the result of that operation arrives back at the client (typically quoted in milliseconds).

System Throughput. The throughput achieved from a particular setup (i.e. system implementation installed on a specific infrastructure) is the number of operations served in a period of time (typically quoted in requests per second).

Data Freshness. Data freshness refers to how long it takes clients connected to a remote DC to be able to read the result of a DB operation that changes state (typically, a WRITE operation) in the local DC. In most implementations, data freshness is traded for throughput and operation latency optimisation i.e. a higher throughput and lower latency are preferred over reading the latest changes from remote DCs.

Causal Consistency. A system is causally-consistent if all operations that are causally-related are seen in the same order across all the nodes in the DDB [7]. Two operations a and b are deemed to be potentially causally-related, denoted by $a \rightarrow b$ (i.e. a leads to b), if at least one of three criteria holds [12]:

1) **Thread of Execution:** Given a process P, operations within the same process are causally-related i.e. $a \rightarrow b$ if P performs a, then it performs b.
2) **Reads from:** Given a WRITE operation a, if b reads the result of a, then $a \rightarrow b$.
3) **Transitivity:** If $a \rightarrow b$, and $b \rightarrow c$, then $a \rightarrow c$.

Conversely, the order of *concurrent* operations across the different nodes is not guaranteed. Concurrent operations are those that are not related through causality, and are therefore essentially unrelated with respect to the data items read and/or written [7]. Two operations a and b are deemed to be concurrent if $a \nrightarrow b$ and $b \nrightarrow a$, and so can be replicated in any order across the DDB cluster, without violating CC.

Conflict Handling. In order for a DDB to support high availability with low latency, WRITEs need to be accepted at any node without requiring co-ordination, at least in the critical path, with other nodes [9]. A state of conflict causes consistency between different nodes to be broken. A conflict is declared when the same data element in two non-colocated replica gets updated concurrently. Two operations on the same data element are conflicting if they write a different value, and are not related by causality [7] i.e. they are concurrent. Concurrent and conflicting operations, in the context of a DDB, can therefore be defined as follows:

- Let Θ_1 and Θ_2 define a WRITE operation on a data item identified by key k, in DC_1 and DC_2 respectively.
- Let $\Theta_1 = put(k, v_1)$.
- Let $\Theta_2 = put(k, v_2)$.

$\therefore$ Θ_1 and Θ_2 are concurrent and conflicting operations.

Various approaches for conflict detection and conflict resolution have been put forward [13], [14], [15]. The Last-Writer Wins (LWW) is a popular conflict resolution technique where the most recent update is retained in case of conflict. A database which offers CC as well as conflict detection and resolution, and therefore convergence, is said to provide *causal+ consistency* (CC+) [16].

IV. Literature Review

A. Causally-Consistent Databases

In COPS [16], application clients are co-located with a cluster of servers that store a full replica of the DB. A WRITE operation is placed in a queue and sent to peer DCs, where it is stored if all its dependencies are also stored. Dependencies of a WRITE operation are tracked by a client-side library that tracks a *context identifier*. Within a *context*, dependencies of a WRITE operation are defined as the latest version of all keys that have so far been interacted with, guaranteeing causality. Conflicts are handled using a LWW approach.

COPS-GT [16] extends COPS with support for read-only transactions. Clients can request the values of a set of keys, rather than that of a single key, and the DDBMS returns a causally-consistent snapshot of the requested keys. COPS-GT, like COPS, uses a sequentially-consistent key-value store, but changes the client library, the DB and the semantics of the READ and WRITE operations. Keys are mapped to a set of versions for that key, rather than one value as in COPS. Each version is mapped to a value and a set of dependencies, encoded as pairs of <*k*ey,version>, thus supporting READ and WRITE operations in a transactional context.

Bolt-On [9] describes a custom middleware on top of Cassandra, a commercially-available EC DB with a columnar data model which handles replication. Bolt-On implements explicit causality, offloading dependency tracking to the client. Data items are tagged with a set of tuples, each indicating a process identifier and its monotonically-increasing identifier. The dependencies of a WRITE operation are the versions of the keys read to produce that operation. Each client holds an *interest set*, the keys that it needs to read, and a resolver process maintains a causally-consistent view of the keys within this set by fetching the latest versions of the data items and their dependencies.

GentleRain [17] provides CC over a key-value, multi-versioned, sharded and replicated DB. It depends on a custom replication protocol to propagate WRITEs across DCs. Dependency tracking is efficient, as the only meta-data stored with a WRITE operation is a timestamp and a server identifier. Any READ operation can only access versions of data that are created in the local DC, or versions that have been created in remote DCs and replicated across all DCs. This guarantees causality by ensuring that when reading a version, the items which have led to its creation (i.e. its dependencies) are present in all DCs.

Wren [18] takes a somewhat similar approach to [17], but uses Hybrid Logical Clocks (HLC) [19] to timestamp events in a more reliable manner. Furthermore, Wren implements transactional CC, allowing clients to perform read transactions as well as running multiple WRITE operations atomically.

B. Benchmark Workloads

DBMS query workloads are segmented into two broad modes [20], [21]. Online transactional processing (OLTP) workloads consist of WRITE queries that modify small amounts of data, and READ queries that process a few records and project the majority of the attributes available [22]. In OLTP, short response times are crucial to avoid user frustration and business impact [23]. In contrast, Online analytical processing (OLAP) workloads typically consist of read-only queries that traverse a large amount of records, performing aggregations and projecting a small set of attributes [22].

TABLE I
TPC-C TRANSACTIONS

Transaction	Characteristic	Minimum Percentage of mix
New Order	read-write	45%
Payment	read-write	43%
Order Status	read-only	4%
Delivery	read-write	4%
Stock Level	read-only	4%

The TPC-C [24] workload simulates a DB which models a number of geographically-distributed brick and mortar warehouses, each associated to one or more districts. A number of terminals perform transactions on stock available in each warehouse. A TPC-C workload is characterised by five transactions over nine tables containing synthetic data [25], as summarised in Table I. The benchmark specification also defines that an execution should comprise transactions chosen at random, but the final transaction set should maintain a minimum percentage for each type of transaction. Each transaction consists of a number of queries, as shown in Table II [26]. The workload can be throttled by two parameters. The scale factor (sf) specifies the number of records that are generated within the DB. The scale factor determines the number of warehouses available, which in turn determines the number of records generated in the other tables. Conversely, the number of terminals determines the number of parallel threads that the workload generator spawns to execute concurrent transactions. Hence, larger scale factors imply that TPC-C queries are heavier (e.g. record selections operators applied to larger tables), but possibly less contentious (a larger number of records reduces the probability of contention), whilst a larger number of terminals increases the number of concurrent transactions, and thus the probability of contention. Finally, TPC-C defines a set of tests that should be executed to confirm that the system under test guarantees suitable Atomicity, Consistency, Isolation, and Durability (ACID) properties.

The TPC-E [27] workload simulates a DB used by a financial brokerage firm that stores customer-related information (e.g. accounts, holdings, watch lists), broker information (e.g., trades, trade history), and financial market data (e.g., companies, securities, related news items, last trades). The TPC-E data model consists of 33 tables and twice the number of columns when compared to TPC-C, and is seeded with pseudo-real data based on the U.S. and Canada census from 2000, as well as census data and actual listings on the NYSE and NASDAQ stock exchanges [28]

The Smallbank [29] benchmark simulates a banking application that sustains transactions relative to financial accounts. The access pattern of the workload is skewed to define a small number of "hot" accounts upon which most transactions are executed. By nature of the application, transactions in this benchmark involve small number of records.

V. MIDDLEWARE FOR CAUSAL CONSISTENCY

As shown by our review of the literature, current approaches to CC offer very specific interfaces, depend on custom DBs, and offload data-handling functionality to client applications in a way that makes CC non-trivial to implement and use. The majority of the approaches also deal with a simple key-value data model. Although this data model can be the building block of more complex schemas, application developers are used to richer data model structures.

Thespis [30] is a middleware that provides CC as well as conflict detection and resolution, thus achieving CC+. Data is stored in a database which offers SC and a rich data model with effective querying and reporting capabilities on data generated by OLTP systems, but no support for horizontal scalability (i.e. the underlying database does not offer the possibility to scale out on more than one node out of the box). This fits the description of a RDBMS. Relational databases are widely used in production systems, and offer a richer data model with effective querying and reporting capabilities on data generated by OLTP systems. Hence, although not mandatory for the Thespis approach, the implementation assumes that: a) the main data backing engine is a RDBMS; and b) that the client is an application handling "objects", primarily instances of business-domain models.

Thespis tackles several objectives: it enables the use of CC without requiring major application re-engineering, stores data in a format accessible to other systems that need to consume it (e.g. reporting modules), and considers efficiency such that performance overheads of CC guarantees do not outweigh the benefits of using a DDBMS.

These objectives are tackled through the fusion of a number of concepts, most importantly:

1) **The Actor Model** [31], which organises logic in terms of a hierarchical society of "experts" that communicate together via asynchronous message passing. An actor consists of a) a *mailbox* where incoming messages are queued; b) an actor's *behaviour*, or the behavioural logic that is executed in response to a received message; and c) an actor's *state*, in other words the data stored by the actor at a given point in time. Actors process one message at a time, and exist in the context of Actor Systems [32], where hierarchies can form.
2) **Command Query Responsibility Segregation (CQRS)** [33], a software design pattern that applies the concept of Command Query Separation (CQS) [34] in order to maintain separate data models for READs and for WRITEs.
3) **Event Sourcing (ES)** [35], another pattern where all data changes are captured as a sequence of events that are stored in an event log and that, when applied in order, provide a view of the system state at a particular point in time.

Figure 1 illustrates the Thespis middleware that offers an API allowing two operations, READ and WRITE. All operations employ the Actor model to deal with both concurrency issues.

Firstly, the actor-based implementation ascertains that READs happen concurrently. Secondly, it also ascertains that WRITEs on the same object, and in the same replica, happen in a set sequence. The hierarchical nature of Actor Systems is also exploited to reflect a causally-consistent view of the underlying database. The **Writer Actor** and **Reader Actor** are responsible for storing actor states and retrieving business objects from the underlying DB respectively. The **Replication Actor** is responsible for replicating actor state changes from one replica to the other. The core **Middleware** actor system holds a set of actors which provide a view of the underlying DB to the application. Finally the actor system adopts a "child-per-entity" approach, spawning one Entity Actor per type of business object (e.g. DB table), supervising dedicated Entity Instance Actors for each business object instance. The state of the Entity Instance Actor is made up of two elements: the Entity Instance and the Event Log. The events in the Event Log can be applied to the Entity Instance governed by the Entity Instance Actor to retrieve the latest (causally-consistent) version of the entity.

The middleware snapshots data changes in the DB only when received in all the DCs. WRITEs are captured in the middleware layer and, given a new version of an entity being created by any WRITE operation, a set of events representing the new state, compared to the previous version, are extracted.

Finally, the system incorporates a replication protocol, again founded on the Actor model, which encapsulates two algorithms, one running on the *Originating Server* (i.e. the server where a new event is created), the other on the *Remote Server*, or the server which is receiving an event from an *Originating Server*. Key to the replication protocol, and to enforce causality, is the Stable Version Vector (SVV). The SVV is simply a vector of length M, where M is the number of peer DCs. Each element SVV_{DC} in the vector is the latest observed timestamp from the corresponding peer DC. Specifically, the vector element $SVV_{DC_N}[M]$ denotes the latest timestamp observed from DC M within DC N.

Details of the implementation, performance evaluation and correctness assessment of the Thespis middleware are given in our previous work [30]. Results show that the Thespis approach achieves CC+, availability and partition tolerance. Furthermore, inline with the PACELC theorem [36], Thespis can provide CC, whilst optimising operation latency under normal conditions, as well as tolerating network partitions or node failures.

A. Read-only Transactions

ThespisTRX [37] adds to Thespis the functionality for when multiple entities need to be retrieved from the underlying DB, potentially in multiple operations whilst preserving causality. Our example in the context of a social media application [37] shows a common encounter of Time-To-Check-Time-To-Use (TOCTOU) race conditions using the Thespis API, and underlines the need for this extension.

ThespisTRX builds on the Thespis approach with three main additions. First of all, two new system components are introduced, namely:

1) The **Transaction Coordinator**, which is responsible to track transactions that are running in the local node at any point in time;
2) The **Entity Version Log**, which is responsible to hold versions of entities that may be required by all currently-running transactions, and which can easily be queried.

Both new components are implemented within the middleware, of which an instance exists in each DC.

Secondly, the middleware API is extended to support three new operations. STARTTRAN and ENDTRAN signal the start and end of a read transaction respectively, whilst READTRX is essentially an overload of the standard READ operation that takes an additional parameter *TransactionId.* Thirdly, the logic in the business application is slightly adjusted to signal the start and end of a transaction, and passes the transaction identifier to all read operations.

Thirdly, given this model, the logic in the business application is slightly adjusted too: the business application signals the start and end of a transaction, and adds the transaction identifier to all of its read operations.

Details of the implementation, performance evaluation and correctness assessment of the ThespisTRX extension are discussed in [37]. The results from empirical evaluation show that read latency is similar in both Thespis and ThespisTRX, confirming that READs do not interfere with the WRITEs in ThespisTRX that maintain the Entity Version Log. WRITEs have a slightly higher latency in ThespisTRX, which is expected due to the need to store additional meta data (i.e. Entity Version Log) in order to support transactional reads. Nonetheless, the reduction in throughput varies between 1.8% (for read-heavy workloads) and 4.8% (for write-heavy workloads), and therefore does not prohibit the use of ThespisTRX for the same workloads as Thespis.

B. Distributed Integrity Invariant Preservation

Integrity Invariants are application-specific operation preconditions, or rules that determine whether an operation on a data element should be accepted or not. A DDBMS such as Thespis [30] achieves low latency and high availability by allowing operations to be accepted at any DC, and propagated to other DCs asynchronously. The result of an operation accepted at any replica can be propagated to a remote replica at a time when the operation's pre-condition no longer holds [38], leading to an anomaly in the integrity invariant.

ThespisDIIP [39] is an extension of Thespis that brings distributed integrity invariant preservation (DIIP), over and above the original CC+ guarantees. Focus is given to integrity invariants for data values that must be satisfied according to a Linear Arithmetic Inequality (LAI) constraint [40]. These are a set of problems that involve resource allocation [41], such as operations on bank accounts (integrity invariants define that withdrawals cannot request more than the available funds) and order fulfillment operations (an order can be accepted only if there is enough stock). Although important in real-world applications [42], these types of integrity invariants are not *I-confluent* [43], meaning they cannot be preserved by concurrent transactions without co-ordination.

ThespisDIIP employs data-value partitioning (DVP) [44] and takes a novel approach to achieve DIIP by exploiting the

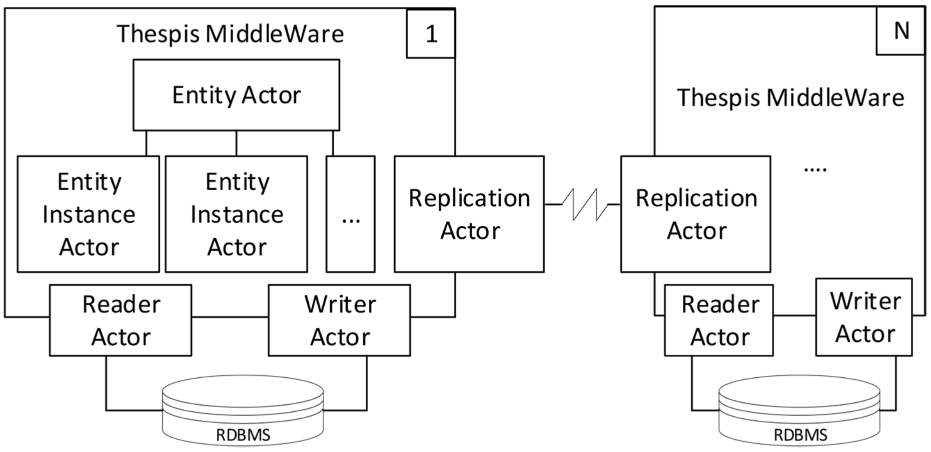

Fig. 1. Thespis Middleware System Model

structure of the underlying RDBMS to trigger background DVP operations when needed.

Design, implementation and benchmarking details are given in [39], where it is shown that the asynchronous operation latency in ThespisDIIP is short enough to eliminate waiting time in the critical path of a typical enterprise application.

VI. Applications to OLTP Workloads

The API offered by the Thespis middleware, as well as its extensions, is well-suited towards atomic workloads, such as YCSB [45], which is used to benchmark Thespis [30] and its extensions [37], [39], but more sophisticated OLTP workloads require richer semantics.

From the list of benchmarks described in our review of the literature, TPC-C is the oldest specification of a transactional workload, however it is also one that is established, and referred to in recent related and orthogonal works in the literature [46], [47]. We thus choose to focus on the TPC-C benchmark in this paper. First, we analyse its queries and subsequently design the necessary semantics to handle them through extensions to Thespis.

Table II summarises the TPC-C transactions and queries. Each of the latter is marked according to a relevant cluster for which semantics are proposed.

A. Primary Key Selection Operations

Queries that return zero or one records based on a predicate on the relation's primary key are trivial and can be satisfied by the Thespis API [30], or any CC DB using a key-value data model. These are marked with † in Table II.

B. Set Selection Operations

Marked with § in Table II, these are queries where the predicate is not based on a primary key, and that thus can return 0 or more records. This is considered feasible to achieve in the Thespis approach, whilst still preserving CC, as shown in Algorithm 1. Given a query Q, first the set of entity instances that match are looked up in the RDBMS. The query is also sent to every Entity Instance Actor that is active in memory. Each Entity Instance Actor returns a message containing a set comprising of one element (the version of the entity instance that the Actor represents) if the version of the entity matches the query. Alternatively, the message represents the empty set. The final result is then the set of entity instances returned from the DB, removing those that are represented by an Entity Instance Actor where an empty result set was returned, and adding those entity instances which are not returned from the DB but are returned from their Entity Instance Actor.

Algorithm 1 Set Selection Query Algorithm

```
Let $S_1 =$ the result of query $Q$ from the RDBMS
Let $S_2$ = the result of query $Q$ executed against the Entity Instance Actors
$result = (S_1 - (\forall s_2 \in S_2$ where $s_2.results = \emptyset) \bigcup ((\forall s_2 \in S_2$ where $s_2.results <> \emptyset))$
```

C. Record Creation Operations

Operations that create a new record are also satisfied by the Thespis API and state-of-the-art CC DBMSs. New records are always created in a causally-consistent context, and therefore the underlying CC+ DBMS is required to manage replication and resolve any potential conflicts. Three queries in the TPC-C benchmark, marked with ¶ in Table II, fall under this segment.

D. Sequence-Management Operations

The query marked with $\odot$ is such an operation, which increments a field with every transaction to always ensure a sequential, monotonically increasing, and unique value.

Although this is managed well by an RDBMS, a special operator is needed by a CC DBMS. The following operator is proposed to satisfy such requirements:

$$\eta : \text{uint64} = \overbrace{\{\text{DC id}\}}^{\text{4 bits}}\overbrace{\{\text{physical time}\}}^{\text{48 bits}}\overbrace{\{\text{logical time}\}}^{\text{12 bits}}$$

Operator η is essentially a hybrid logical clock (HLC) [19] that generates monotonically-increasing values at the microsecond granularity tracking of physical time. The value is prefixed by the unique identifier of the DC where the operation is accepted, essentially allowing the CC DBMS to span across

TABLE II
TPC-C TRANSACTIONS AND QUERIES.
† PRIMARY KEY SELECTION. § SET SELECTION. ¶ RECORD CREATION. ⊙ SEQUENCE MANAGEMENT. ⋇ VALUE INCREMENT. ⊎ VALUE ASSIGNMENT. ⊠ RECORD DELETION. ± JOINS, AGGREGATES ETC.

Transaction	Procedure
New Order	1) Select(whouse-id) from Warehouse † 2) Select(dist-id,whouse-id) from District † 3) Update(dist-id,whouse-id) in District ⊙ 4) Select(customer-id,dist-id,whouse-id) from Customer † 5) Insert into Order ¶ 6) Insert into New-Order ¶ 7) For each item (10 items): a) Select(item-id) from Item † b) Select(item-id,whouse-id) from Stock † c) Update(stock-id,whouse-id) in Stock ⋇ d) Insert into Order-Line ¶
Payment	1) Select(whouse-id) from Warehouse † 2) Select(dist-id,whouse-id) from District † 3) a) Case 1: Select(customer-id,dist -id,whouse-id) from Customer † OR b) Case 2: Non-Unique Select(customer-name,dist-id,whouse-id) from Customer § 4) Update(whouse-id) in Warehouse ⋇ 5) Update(dist-id,whouse-id) in District ⋇ 6) Update(customer-id,dist-id,whouse-id) in Customer ⊎ 7) Insert into History ¶
Order Status	1) a) Case 1: Select(customer-id, dist-id,whouse-id) from Customer † b) Case 2: Non-Unique- Select (customer-name, dist-id,whouseid) from Customer § 2) Select (Max (order-id) ,customer-id) from Order ± 3) For each item in the order: a) Select (order-id) from Order-Line †
Delivery	1) For each district within the warehouse (i.e. ten times): a) Select (no-o-id) from New-Order † b) Delete(order-id) from New-Order ⊠ c) Select (customer-id) from Order † d) Update(order-id) from Order ⊎ e) For each item in the order (i.e. ten times): i) Update (delivery-date) from Order-Line ⊎ f) Select (Sum (amount)) from Order-Line ± g) Update(balance) from Customer ⋇ h) Update(delivery-cnt) from Customer ⋇
Stock Level	1) Select (d-next-o-id) from District † 2) Select count(distinct (s-i-id)) from OrderLine,Stock ±

a maximum of 16 DCs, which is deemed sufficient for our use cases. Finally, reserving 12 bits for the logical time part of the HLC, accepting a maximum value of 4096, is also deemed sufficient, based on results that show that this rarely exceeds 100 [19] even in the worst case of clock skew.

This operator satisfies two of the three requirements of the TPC-C workload: it generates a monotonically increasing and unique value at any point in time however, the values are not guaranteed to be sequential. This latter guarantee could be achieved through co-ordination between the DDBMSs nodes, resulting however in the loss of high availability and throughput of the DDBMS. Thus, we see the η operator being useful in the CC DBMS, where application semantics can relax the sequential constraint.

Another approach is a UID generator [48], where part of the acceptable range of values in the sequence is allocated to each DC, that in turn can generate sequential values from its allocation. This is a simpler approach however it assumes that the number of DCs is static and known beforehand, and does not support a cluster that can shrink or grow to the needs of the application.

E. Value-Increment Operations

Trivially, we define value-increment operations as those operations that increment the value of a field in a tuple, such as those marked with ⋇ in Table II. A special operator is also required by a CC DBMS to handle such operations. The following operator is proposed to satisfy such requirements:

$$\gamma(t) : \text{int64} = \sum_{0}^{n-1}[v_0, v_2, v_3, ..., v_{n-1}]$$

Essentially, the operator $\gamma(x)$ yields a *Grow-Only Counter* [49], that stores a value v for every DC participating in the cluster. Executing $\gamma(t)$ at DC x increments the value v_x by t.

Grow-Only Counters are conflict free replicated data types (CRDTs) that support increment operations in a distributed environment without co-ordination [49]. Therefore, they allow a CC DDBMS to execute such operations safely, without introducing co-ordination and losing on high availability and performance.

F. Value-Assignment Operations

Value-assignment operations set an absolute value of a field in a tuple. Such queries, marked with $\uplus$ in Table II, are equivalent to WRITE operations in a CC DDBMS and do not strictly require any special semantics. However, some OLTP workloads may need to enforce stronger consistency on such operations in order to avoid lost updates, specifically the effects of concurrent operations which are subsequently resolved via the LWW strategy by the CC DDBMS. Such consistency guarantees have been shown to be incompatible with a distributed, highly-available environment [50].

Thespis (with the ThespisDIIP extension [39]) already supports application-specific configuration to enforce invariants that satisfy LAI constraints. We extend this configuration further in order to define fields that require a strong consistency, and therefore co-ordination between nodes. This is in-line with other approaches to multi-level consistency in the literature [51] [52] and we believe that it is an important feature that makes a CC DDBMS applicable to common OLTP workloads where, for some operations, strong consistency is preferred over operation latency or high availability.

G. Record Deletion Operations

Operations that delete records, such as the one marked with $\boxtimes$ in the Delivery transaction, are achievable by the same semantics as a WRITE operation of the Thespis API, and therefore inherit the same causal consistency guarantees.

H. Operations involving Joins, Aggregates, Sorting etc.

OLTP queries may comprise other relational algebra operations, such as joins, aggregates and sorting. Such queries in the TPC-C workload are marked with $\pm$ in the Stock Level and Order Status transactions.

Join operations can be considered equivalent to a Cartesian product of two relations, followed by a set selection operation [53] and are therefore achievable with the same semantics. Similarly, aggregates and sorting can be implemented in the actor system using a combination of set selection operation semantics and corresponding aggregate or sorting logic. Building on the semantics of operations that have already been defined ensures that CC is guaranteed.

VII. Discussion

The original REST interface of Thespis allows the execution of workloads using a key-value data model. However, we have now shown that the Thespis approach scales to handle also richer data models, such as the relational data model, given that the operators discussed in Section VI are made available in the Thespis API.

With these operators, our analysis shows that it is possible to satisfy the semantics of all the TPC-C transactions in Thespis. Specifically, a client application can interface with Thespis to execute the TPC-C workload in a CC+ distributed environment, thus benefiting from the scalability and high availability properties of a CC+ DBMS.

However, it is important to highlight that Thespis still remains a distributed CC+ DDBMS and therefore foregoes a number of guarantees that a SC RDBMS offers, and that the TPC-C benchmark requires. Most importantly, being a distributed CC+ DBMS, Thespis does not conform with the ACID guarantees that the official TPC-C specification mandates.

The first area of divergence relates to atomicity guarantees. Given that Thespis only supports read-only transactions, via the ThespisTRX extension, multiple WRITE operations are always treated distinctly. Therefore, Thespis does not provide full support for the atomicity criteria that the TPC-C benchmark requires for its transactions.

Furthermore, Thespis guarantees CC, the strongest level of consistency that a DDB can guarantee whilst also supporting high availability. As expected, this level of consistency is not sufficient to guarantee the consistency requirements of TPC-C, which is expected and in-line with orthogonal literature [46].

The isolation requirements of TPC-C are also not adhered to under Thespis, where there is no notion of transactions. A higher level of conformity is guaranteed through the ThespisTRX extension, which protects read-only transactions from phenomena such as phantom and non-repeatable reads. However, the lack of support for WRITE transactions does not prohibit other phenomena such as dirty writes.

Lastly, being a DDB that can accept WRITE operations at multiple DCs, Thespis does not satisfy the durability requirements of TPC-C. Specifically, any WRITE operation (such as record creation, value assignment and record deletion operations) can be overwritten by the LWW conflict resolution operation. Given the possible occurrence of conflicts, and the fact that these cannot be prohibited without losing high availability, the effects of a WRITE operation cannot be deemed sufficiently durable in Thespis to conform with the requirements of the TPC-C benchmark.

VIII. Conclusions

The problem domain is well studied and different approaches have been proposed, including domain specific languages [54], data types that support various consistency levels [55], and instructions expressed in the application's third-generation language [48], [56].

We tackle the problem by proposing a middleware that scales an RDBMS into a CC+ DDBMS, and sits behind an API that is accessible to application developers through an easy interface, abstracting the complexities of CC.

An important contribution of this paper is the analysis of the TPC-C benchmark, a popular OLTP workload, and the proposal of further extensions to the Thespis API that increase the scope of the operations permitted by the CC middleware. We also innovatively show how these extensions allow execution of the OLTP queries in the context of a CC DDBMS whilst retaining an easy-to-use API. We also further analyse the TPC-C workload and highlight areas where Thespis suffers from the intrinsic properties of a CC+ DDB, and therefore does not adhere to the requirements of the TPC-C specification. Nonetheless, given that CC is deemed as a sufficiently strong consistency for enterprise applications, a CC+ DDBMS remains an important tool for such problem domains that are able to forego ACID guarantees for other desirable properties such as scalability and high availability.

Finally, future work consists of the implementation of the semantics proposed in this paper, as well as the execution of suitable benchmarks and a discussion of their respective results. A similar analysis of other transactional workloads, such as the ones mentioned in our review of the literature, is also a future direction of research.

Acknowledgment

This work is partly funded by the ENDEAVOUR Scholarship Scheme (Malta), part-financed by the European Union – European Social Fund ESF under Operational Programme II – Cohesion Policy 2014-2020.

References

[1] E. A. Brewer, "Towards robust distributed systems," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00, vol. 7, 2000. doi: 10.1145/343477.343502. ISBN 1581131836

[2] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002. doi: 10.1145/564585.564601

[3] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, July 1990. doi: 10.1145/78969.78972

[4] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, January 2009. doi: 10.1145/1435417.1435432

[5] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998. doi: 10.1145/279227.279229

[6] M. M. Elbushra and J. Lindström, "Eventual consistent databases: State of the art," *Open Journal of Databases (OJDB)*, vol. 1, no. 1, pp. 26–41, January 2014.

[7] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995. doi: 10.1007/bf01784241

[8] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, and convergence," *University of Texas at Austin Tech Report*, vol. 11, 2011.

[9] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013. doi: 10.1145/2463676.2465279 pp. 761–772.

[10] K. Spirovska, D. Didona, and W. Zwaenepoel, "Optimistic causal consistency for geo-replicated key-value stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 527–542, March 2021. doi: 10.1109/tpds.2020.3026778

[11] S. Braun, A. Bieniusa, and F. Elberzhager, "Advanced domain-driven design for consistency in distributed data-intensive systems," in *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, April 2021. doi: 10.1145/3447865.3457969 pp. 1–12.

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. doi: 10.1145/359545.359563

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, August 2013. doi: 10.1145/2491245

[14] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 172–182, December 1995. doi: 10.1145/224057.224070

[15] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, April 2010. doi: 10.1145/1773912.1773922

[16] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. Association for Computing Machinery (ACM), 2011. doi: 10.1145/2043556.2043593. ISBN 9781450309776 pp. 401–416.

[17] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, November 2014. doi: 10.1145/2670979.2670983 pp. 1–13.

[18] K. Spirovska, D. Didona, and W. Zwaenepoel, "Wren: Nonblocking reads in a partitioned transactional causally consistent data store," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, June 2018. doi: 10.1109/dsn.2018.00014 pp. 1–12.

[19] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Lecture Notes in Computer Science*, M. K. Aguilera, L. Querzoni, and M. Shapiro, Eds. Springer International Publishing, 2014. doi: 10.1007/978-3-319-14472-6_2 pp. 17–32.

[20] S. Elnaffar, P. Martin, and R. Horman, "Automatically classifying database workloads," in *Proceedings of the eleventh international conference on Information and knowledge management - CIKM '02*. ACM Press, 2002. doi: 10.1145/584792.584898 pp. 622–624.

[21] L. Li, G. Wu, G. Wang, and Y. Yuan, "Accelerating hybrid transactional/analytical processing using consistent dual-snapshot," in *International Conference on Database Systems for Advanced Applications*. Springer International Publishing, 2019. doi: 10.1007/978-3-030-18576-3_4 pp. 52–69.

[22] M. Bach and A. Werner, "Hybrid column/row-oriented DBMS," in *Advances in Intelligent Systems and Computing*. Springer International Publishing, September 2015, pp. 697–707. doi: 10.1007/978-3-319-23437-3_60

[23] N. Poggi, D. Carrera, R. Gavalda, E. Ayguadé, and J. Torres, "A methodology for the evaluation of high response time on e-commerce users and sales," *Information Systems Frontiers*, vol. 16, no. 5, pp. 867–885, October 2014. doi: 10.1007/s10796-012-9387-4

[24] F. Raab, "TPC-C - the standard benchmark for online transaction processing (OLTP)," in *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*, 1993.

[25] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki, "From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored," in *Proceedings of the 16th International Conference on Extending Database Technology - EDBT '13*, 2013. doi: 10.1145/2452376.2452380 pp. 17–28.

[26] S. T. Leutenegger and D. Dias, "A modeling study of the TPC-c benchmark," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 22–31, June 1993. doi: 10.1145/170036.170042

[27] T. P. P. C. TPC, "Tpc benchmark e," 2010.

[28] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica, "TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study," *ACM SIGMOD Record*, vol. 39, no. 3, pp. 5–10, February 2011. doi: 10.1145/1942776.1942778

[29] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," *ACM Transactions on Database Systems*, vol. 34, no. 4, pp. 1–42, December 2009. doi: 10.1145/1620585.1620587

[30] C. Camilleri, J. G. Vella, and V. Nezval, "Thespis: Actor-Based Causal Consistency," in *Database and Expert Systems Applications (DEXA), 2017. 28th International Workshop on Big Data Management in Cloud Systems*. IEEE, August 2017. doi: 10.1109/dexa.2017.25 pp. 42–46.

[31] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[32] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986. doi: 10.7551/mitpress/1086.001.0001

[33] G. Young, "CQRS documents by Greg Young," 2010. [Online]. Available: https://github.com/keyvanakbary/cqrs-documents

[34] B. Meyer, *Eiffel: The Language*. Prentice-Hall, Inc., December 1992. ISBN 0-13-247925-7. doi: 10.1016/0950-5849(92)90131-8

[35] M. Fowler, "Event sourcing," December 2005. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html

[36] D. Abadi, “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story,” *Computer*, vol. 45, no. 2, pp. 37–42, February 2012. doi: 10.1109/mc.2012.33

[37] C. Camilleri, J. G. Vella, and V. Nezval, “ThespisTRX: Causally-consistent read transactions,” *International Journal of Information Technology and Web Engineering (IJITWE)*, vol. 15, no. 1, pp. 1–16, January 2020. doi: 10.4018/ijitwe.2020010101

[38] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, and N. Preguiça, “IPA: Invariant-preserving applications for weakly-consistent replicated databases,” *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 404–418, December 2018. doi: 10.14778/3297753.3297760

[39] C. Camilleri, J. G. Vella, and V. Nezval, “ThespisDIIP: Distributed integrity invariant preservation,” in *International Conference on Database and Expert Systems Applications*. Springer International Publishing, 2018. doi: 10.1007/978-3-319-99133-7_2 pp. 21–37.

[40] D. Barbará-Millá and H. Garcia-Molina, “The demarcation protocol: A technique for maintaining constraints in distributed database systems,” *The VLDB Journal - The International Journal on Very Large Data Bases*, vol. 3, no. 3, pp. 325–353, July 1994. doi: 10.1007/bf01232643

[41] N. Krishnakumar and A. J. Bernstein, “High throughput escrow algorithms for replicated databases,” ser. VLDB ’92. Morgan Kaufmann Publishers Inc., 1992. ISBN 1558601511 pp. 175–186.

[42] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Feral concurrency control: An empirical investigation of modern application integrity,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, May 2015. doi: 10.1145/2723372.2737784 pp. 1327–1342.

[43] P. Bailis, A. Fekete, M. J. Franklin, and A. Ghodsi, “Coordination avoidance in database systems,” *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 185–196, 2014. doi: 10.14778/2735508.2735509

[44] N. Soparkar and A. Silberschatz, “Data-valued partitioning and virtual messages,” in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '90*. ACM Press, 1990. doi: 10.1145/298514.298587 pp. 357–367.

[45] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. ACM Press, 2010. doi: 10.1145/1807128.1807152 pp. 143–154.

[46] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan, “CLOTHO: directed test generation for weakly consistent database systems,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, October 2019. doi: 10.1145/3360543

[47] A. Chikhaoui, K. Boukhalfa, and J. Boukhobza, “A cost model for hybrid storage systems in a cloud federations,” in *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, ser. Annals of Computer Science and Information Systems, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds., vol. 15. IEEE, September 2018. doi: 10.15439/2018F237 pp. 1025–1034.

[48] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, “Putting consistency back into eventual consistency,” in *Proceedings of the Tenth European Conference on Computer Systems*, April 2015. doi: 10.1145/2741948.2741972 pp. 1–16.

[49] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of convergent and commutative replicated data types,” Ph.D. dissertation, Inria–Centre Paris-Rocquencourt; INRIA, 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

[50] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Highly available transactions: Virtues and limitations,” *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 181–192, November 2013. doi: 10.14778/2732232.2732237

[51] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, “Making geo-replicated systems fast as possible, consistent when necessary,” in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI’12)*, 2012. ISBN 9781931971966 pp. 265–278.

[52] A. Bouajjani, C. Enea, M. Mukund, G. Shenoy, and S. Suresh, “Formalizing and checking multilevel consistency,” in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer International Publishing, 2020. doi: 10.1007/978-3-030-39322-9_18 pp. 379–400.

[53] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Pearson, June 2015. ISBN 0133970779

[54] M. Milano and A. C. Myers, “Mixt: A language for mixing consistency in geodistributed transactions,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 53, no. 4. ACM, June 2018. doi: 10.1145/3192366.3192375 pp. 226–241.

[55] B. Holt, J. Bornholt, I. Zhang, D. Ports, M. Oskin, and L. Ceze, “Disciplined inconsistency with consistency types,” in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, October 2016. doi: 10.1145/2987550.2987559 pp. 279–293.

[56] M. Köhler, N. Eskandani, P. Weisenburger, A. Margara, and G. Salvaneschi, “Rethinking safe consistency in distributed object-oriented programming,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, November 2020. doi: 10.1145/3428256