Typhon: Consistency Semantics for Multi-Representation Data Processing

Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi
Department of Computer Science, University of California, Santa Barbara
{vaibhavarora, nawab, agrawal, amr}@cs.ucsb.edu

***Abstract*—Variety of data has led to the fall of the "One size fits all" paradigm in databases. Applications now store data in multiple data representations, rather than just storing the data in a single relational database. However, applications only provide consistency semantics at the boundaries of a single datastore, corresponding to specific representation. This can lead to semantically inconsistent executions, which can have undesirable consequences for the application users. We propose, *Typhon,* a multi-representation data processing framework, which defines consistency semantics across multiple representations. Typhon introduces the notion of *entities*, which link the data present in diverse representations, and defines implicit causal guarantees, which capture the logical order intended by the application user. Typhon lays the foundation of a consistency model for multi-representation data, over which applications can build their protocols and reason about the consistency semantics of their existing solutions. We then propose a protocol, *Cerberus,* which provides implicit causal guarantees defined by Typhon's multi-representation consistency framework. Our evaluations show that Cerberus's performance is close to the solution providing no consistency guarantees across multiple representations, and outperforms baseline locking protocols providing Typhon's consistency guarantees across multiple data representations.**

***Keywords*-Data Variety, Consistency, Transactions, Causal Consistency, Multiple data representations**

I. Introduction

One of the most challenging aspects of current data management ecosystem is the *variety* of the data. The variety of data has led to the fall of the "one size fits all" paradigm [28], and applications now store data in different systems to handle diverse workload and application characteristics, rather than just using a single RDBMS. Data objects stored across these different systems have semantic relationships between them. However, typically applications guarantee consistency semantics only at the granularity of individual data-storage systems. This can lead to inconsistent semantics for user applications that encompass multiple data sources.

To illustrate inconsistent semantics in applications accessing data across multiple representations, consider a social networking application. The social network consists of users, with attributes and a list of friends. The user attributes are stored in a key-value store and friend relationships are stored in a graph database, to take advantage of the different characteristics of user attribute data and the friendship graph. Now consider two users: Alice and Bob. The specifications of the application only permit the friends of a user to access a user's personal information stored in the key-value store. For example, since Alice and Bob are friends, Bob has access to Alice's phone number. Suppose, Alice wants to change her phone number and does not want Bob to access it. Alice deletes Bob from her friends-list and subsequently, she changes her phone number. Any request by Bob to read Alice's phone number should not be able to access Alice's new phone number. Providing even the strongest guarantees individually at the key-value store and the graph database does not provide this consistency guarantee across the different systems (Section II). Enforcing consistency semantics at the granularity of each representation might provide performance benefits for the application, as no coordination is needed across representations. However, this consistency model is difficult to comprehend for the end user of the application and can have undesirable consequences in certain situations.

Apart from different workload characteristics, another reason for storing data in multiple representations is the need for different access permissions and diverse origins of the data. Some applications need to access multiple data sources, managed by different departments within an organization, with different access protocols. Consider the following example of medical data spread across representations. One data representation stores metadata related to a patient, like name, age, address etc. Another representation stores the list of doctors who have access to patient data. This representation can only be accessed by the hospital and the patient. A third data representation stores prescriptions and lab results, which are only accessible to the patient and the doctors permitted in the second representation. Similar to the social network scenario, if the consistency guarantees are only provided at the granularity of individual representation, it can lead to semantically inconsistent executions. For example, we can have scenarios where a doctor deleted by a patient is able to add prescriptions for the patient. To avoid these application anomalies, we need to define consistency guarantees for data spread across multiple representations.

A possible solution would be to use external consistency [10] to order the operations across representations. But, enforcement of external consistency is an expensive proposition and overly restrictive, since it imposes an order among *all* operations across all given representations. Google Spanner [10] uses atomic clocks to provide external consistency. These clocks are expensive and are not widely available. Another mechanism would be to use deterministic schemes [29], which employ a sequencer for ordering all operations across representations. This is also expensive, and restricts the concurrency of operations at each representation. There is a need for solutions which can provide coordination among multiple data representations at a finer granularity. Furthermore, formal consistency models are needed to reason about the different techniques providing consistency across multiple representations of data.

We propose *Typhon,* a *multi-representation data processing framework*, which defines consistency semantics over data stored and accessed in multiple representations. The data model in Typhon allows the application developer to express dependencies across different application data by defining logical *entities* to link data across representations. Each entity semantically links different *data items*. In the social network described above, the user tuple in the key-value store and the user node in the graph represent two different data items related to the same user entity. Defining a user as an entity enables Typhon to express data consistency semantics at the granularity of a user. By relating the operations on items belonging to the same logical entity, Typhon can capture dependencies across representations. Typhon's consistency model defines *implicit causal dependencies*, which capture the logical order intended by the application user.

We then present a protocol, *Cerberus,* which operates in Typhon's framework and provides consistency semantics for operations accessing data across representations. Cerberus targets social networking and other similar applications, where operations access data items related to a single entity, across multiple representations. Cerberus employs a *single-entity transaction model*, which allows each transaction to access multiple items related to a single entity, across multiple representations. Cerberus uses a lightweight mechanism based on version vectors to ensure that all the application requests satisfy implicit causal dependencies for the entity accessed in a transaction. Our key contributions can be summarized as:

- We propose Typhon, which introduces a novel consistency model for accessing data spread across multiple representations. Typhon presents a formal model which can be used to reason about consistency guarantees provided by protocols in a multi-representation setting.
- We develop a protocol, Cerberus, which ensures that every operation satisfies implicit causal dependencies, which capture the logical order intended by application users, across different representations of data.
- Cerberus adopts a single-entity transaction model, which enables providing transaction guarantees only at the fine granularity of an entity, without requiring a general-purpose transaction model.
- Evaluation results show that Cerberus provides a lightweight and scalable mechanism, with relatively small overhead when compared to a solution providing no consistency guarantees for operations accessing data across representations.

Formal description of Typhon's consistency model is presented in Section II. Section III gives a detailed description of the single-entity transaction model and the Cerberus protocol. Section IV discusses the scaling and recovery mechanism in Cerberus. The evaluation results for Cerberus are presented in Section V. Section VI discusses the related work and we conclude in Section VII.

II. Typhon's Multi-Representation Consistency Model

Application data in Typhon is stored in *n representations*, R_1, $R_2 \ldots R_n$. The database consists of *m entities*. Each entity may have information stored in different representations, in the form of *data items*. x_j refers to a data item corresponding to entity x in representation R_j. An entity may not have a data item in a particular representation, which implies that x_j can be null.

The data in different representations is accessed by transactions comprising reads and writes over a subset of data items. Each transaction, T_i consists of a begin, b_i, set of reads, $r_i(x_j)$, and writes, $w_i(x_j)$, followed by a commit, c_i. The operations of a transaction can span data items present in different representations. We assume each representation either provides serializable transactions or provides atomic read and write guarantees over the individual items. A *history* is defined as a schedule of begin, read, write and commit operations on items across all representations. The relation $<$ between operations implies their order in the history.

Typhon models the dependencies between transactions at the granularity of the entities. This allows us to semantically link data items present at different representations. The dependencies can be divided into dependencies within a representation and those across representations. At a particular representation, any two operations accessing a data item, where one of the operations is a write, are defined to have a *conflict* and have an order among them. Additionally, we also define dependencies between transactions accessing data items related to the same entity, across representations. These dependencies capture the implicit order of operations on the same entity.

Conflict Graph. A conflict graph [6] captures the dependencies of a transaction on other transactions in a given history. If a conflict graph for a particular history is acyclic, the given history is serializable [6]. We now define the conditions for constructing the *conflict graph* in Typhon's multi-representational consistency model. We will then illustrate that the inconsistent semantic execution in the social network scenario described in Section I, cannot be captured in the traditional conflict graph model. We will also show that by adding an additional class of edges, Typhon's multi-representational consistency model is able to capture the inconsistent execution.

Transactions are the nodes in the conflict graph. Transactions accessing the same data item at a representation can lead to $w-w$, $w-r$ and $r-w$ edges. The conditions for adding these edges are the same as the conditions for adding such edges in the traditional conflict graph. There exists an edge from T_i to T_j, $T_i \rightarrow T_j$, if transactions T_i and T_j access the same data item x_k, and T_i precedes and conflicts with one of T_j's operations. $w-w$, $w-r$ and $r-w$ edges reflect the writes-after, reads-from and reads-before relationship between two transactions relative to a data item.

Implicit Causal Edges: Transactions accessing the same entity across representations can have implicit causal edges between them. These edges are added to capture *implicit causal dependencies*, which arise on different items related to a single entity, but spread across representations. The conditions for adding implicit causal edges are below.

Definition 1: There exists an implicit causal edge from T_i to T_j, $T_i \rightarrow T_j$, if transactions T_i and T_j access data items x_k and x_l of the same entity x, and $o_i(x_k) \in T_i$ and $o_j(x_l) \in T_j$ and $c_i < b_j$, where $o_i(x_k)$ and $o_j(x_l)$ are read or write operations

and one of $o_i(x_k)$ or $o_j(x_l)$ is a write. T_i and T_j are defined to have an implicit causal dependency between them.

The implicit causal edges capture the dependencies between transactions at the granularity of an entity. Typhon defines dependencies between transactions that have a "happens-before" relationship and have conflicting operations on a particular entity. If a transaction T accesses an entity, it is defined to be implicitly causally dependent on any transaction which committed before transaction T began, and has a conflicting operation on the corresponding entity. *The implicit causal dependency captures all the transactions that potentially might have affected transaction T.*

A cycle in the conflict graph, with one of the edges being an implicit causal edge, indicates that "happens-before" relationships have not been preserved. Hence, the addition of the implicit causal edges to the conflict graph model can be used to detect inconsistent semantic executions.

Inconsistency Example Description. We formally analyze the example of an inconsistent semantic execution order described in Section I. Suppose, x is Alice's entity and y is Bob's entity. Key-value store and graph representations are represented as R_1 and R_2 respectively. x_1 refers to the data item storing Alice's information in the key-value store and x_2 is the data item storing Alice's information in the graph representation. Operations for removing Bob from Alice's friend-list, modifying Alice's phone number and accessing Alice's phone number are depicted by 3 transactions, T_1, T_2 and T_3 respectively.

$T_1 = b_1 w_1(x_2) c_1$ //Alice removes Bob from her friend-list
$T_2 = b_2 w_2(x_1) c_2$ //Alice changes her phone number
$T_3 = b_3 r_3(x_2) r_3(x_1) c_3$ //Bob reading Alice's phone number

In transaction T_1, deletion of Alice's friendship with Bob leads to modification of the edge between Alice's and Bob's node in the graph representation. As friend relationships are bi-directional, the deletion of Bob from Alice's list would also lead to a write on the item y_2, Bob's friendship information, but we omit it for brevity. Reading Alice's phone number in T_3, comprises a check to see if Bob is still a friend of Alice and if he is, then reading her phone number. T_3 is executed at corresponding representations. T_1 and T_3 arrive concurrently and T_2 starts after T_1 commits. Both representations guarantee atomic execution. Consider the execution of the semantically inconsistent history, H_1.

H_1: $b_1 b_3 r_3(x_2) w_1(x_2) c_1 b_2 w_2(x_1) c_2 r_3(x_1) c_3$

This execution results in a scenario where T_3 reads the older version of Alice's friend-list, before she deleted Bob. T_3 also reads her new phone number. This execution will lead to Bob reading the new phone number of Alice, which should not be allowed as Alice and Bob were not friends when she changed her phone number.

Consider the conflict graph of the history H_1 in Typhon's multi-representational consistency model, shown in Figure 1. Suppose, T_0 is a transaction initializing x_1 and x_2. First, consider the edges arising only out of conflicts at individual representations. Such edges are represented by solid edges in the figure. The graph is acyclic, when considering only these

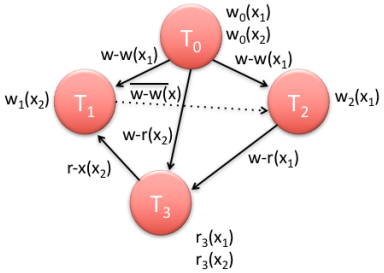

Fig. 1: Conflict Graph showing inconsistent semantics

edges, and the execution of this history is permissible by the conflict graph. This shows that only considering the traditional conflict graph edges is not sufficient to capture inconsistent execution order over the data spread across representations.

Now, we add the implicit causal write-write edge, $\overline{w-w}$, arising from the implicit causal order between the writes on x_2 and x_1, performed by transactions T_1 and T_2 respectively. T_1 and T_2 access items related to entity x, $w_1(x_2) \in T_1$ and $w_2(x_1) \in T_2$, and $c_1 < b_2$ in H_1. Hence, from Definition 1, there is an implicit causal write-write edge, $\overline{w-w}$ from $T_1 \rightarrow T_2$. This edge is represented by the dashed line in the figure. The conflict graph has a cycle when this implicit causal edge is considered, because T_3's read operations do not obey the causal order between T_1 and T_2. For any particular execution, the presence of a cycle in Typhon's multi-representation conflict graph, with one of the edges being an implicit causal edge, indicates that the "happens-before" relationships have been violated. Analogous to a traditional conflict graph, Typhon's multi-representation conflict graph, can be used as a tool to reason about the consistency guarantees for applications storing and accessing data at multiple representations.

III. The Cerberus Protocol

We now design a protocol, Cerberus, which is targeted to social network and other such web applications, and preserves the implicit causal dependencies defined in Typhon. We first describe the application characteristics and the execution model suited for these applications, then describe Cerberus.

A. Application Characteristics

Social media and many other web-applications store their data in datastores that provide atomic guarantees only for a single key [4], [8], [1]. Some example of such datastores are graph stores like Tao [8], key-value stores like Voldemart [4], and document databases like Couchbase [1]. Such datastores are chosen since atomic operations are needed at the granularity of a single logical user entity. Such applications also have their data spread across many representations, due to different performance characteristics of representations, varied access permissions and diverse origins of datasets. These datastores provide only atomic read and atomic read-modify-write guarantees at a single data representation. However, these applications also need order-preserving guarantees over operations on data related to a logical entity, but spread over different representations. This implies that if there are conflicting non-concurrent operations on data items related to an entity in a particular order, all the operations should observe the order, even across representations. In a social network, if a user deletes a friend first and then changes their

phone number, all application requests should observe that order. An execution model and protocol only providing atomic guarantees at the granularity of a single item is not sufficient.

B. Single Entity Transaction Model

Based on the application characteristics, we need an execution model which provides applications the ability to execute operations at the granularity of a single logical entity, and not only a single data item. To target the application scenarios described, Cerberus employs a *single-entity transaction model*. Clients access and modify data as transactions. Transactional boundaries are only needed at the granularity of a single entity. Hence, transactions in Cerberus only access a single entity. Writes are restricted to a single representation, as such applications typically require atomic writes to a single representation based on the information in other representations, like writing a post on a users page based on the friendship status. The read operations access data items related to the same entity at different representations and the write operation updates the entity at a particular representation. A transaction executes in two phases. In the first phase, an application client performs all the reads. Writes, validation and commit are performed in the second phase. A read-only transaction can complete in a single phase. The single-entity transactional model allows the application to express user operations relating multiple data items present in different representations, without using a general-purpose transactional model. Protocols can be designed in such an execution model to provide guarantees only at the fine granularity of an entity.

C. Cerberus Overview

Cerberus ensures that if there are any two transactions which are non-concurrent and have conflicting operations on the same entity, the order among these two transactions would be preserved even across representations.

The system architecture for employing Cerberus in Typhon's multi representational framework is illustrated in Figure 2. Cerberus uses a middleware approach, which has been widely used in partitioned and replicated environments [27]. In this approach, client requests are sent to a middleware layer, which is the Operation processing layer (OPL) in Cerberus's case. The OPL then routes requests to the corresponding data storage location(s) and collects the results and sends them back to the client. OPL also maintains metadata related to the data items and representations.

Each representation consists of a datastore layer which provides the isolation guarantees at the representation and the Cerberus layer, which preserves implicit causal order across representations. The protocol executes as a thin layer over the isolation guarantees provided at each representation. Each representation provides a key-value datastore interface, which supports single-key atomic read and test-and-set operations. Executing as a thin layer over the isolation scheme at each representation provides Cerberus the flexibility to augment existing datastores. We envisage the entities to be logical connections among the data at different representations, like a user for a social networking application. We believe that the entity identification should be done by the application developer, based on the application semantics.

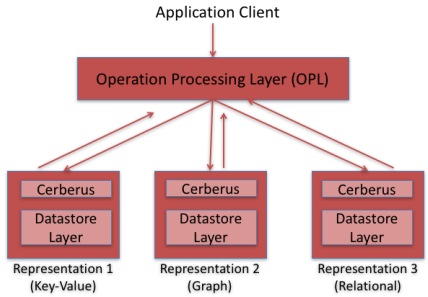

Fig. 2: System architecture for Cerberus

Entity	Data Items	Global Version Vector (GVV)
x	$\{x_1\ x_2, x_3\}$	[0 1 3]
y	$\{y_1\ y_2, y_3\}$	[1 4 2]

Fig. 3: Global Metadata in Cerberus

D. Metadata Description

Each data item has a *version number* associated with it, represented as v_{x_i}, for data item x_i. The OPL maintains information about the data items associated with an entity and the latest version of these data items. The OPL maintains a *version vector*, that comprises the latest version number of all the data items associated to the entity. This global version metadata is referred to as the *global version vector* (GVV) of an entity. The GVV of an entity x comprises $v_{x_1}, v_{x_2} ... v_{x_n}$, where $x_1, x_2 \ldots x_n$ are the data items of entity x and n is the number of representations. $R_1 \ldots R_n$ are the corresponding representations. Figure 3 illustrates the GVV for two entities; each entity has data items associated with it at three representations.

Representations also store some metadata. Each representation maintains two local version vectors, an *update version vector* (UVV) and a *read version vector* (RVV) associated with each data item. The *UVV* of x_i stores a version vector corresponding to entity x. The i^{th} entry of x_i's UVV, represents the latest version of data item, x_i. All other entries of x_i's UVV represent the latest version of the data items, $x_1..x_n$, known at the representation R_i, at the time of x_i's last update. The *RVV* of x_i also stores a version vector corresponding to entity x. All the entries of x_i's RVV represent the latest version of the data items, $x_1..x_n$, known at R_i, at the time of x_i's last read. Figure 4 illustrates the local version vectors at a representation, R_1, for data items x_1 and y_1. All the entries in the version vectors are initialized to 0 at start-up and monotonically increase as updates and reads are processed.

E. Transaction Execution

Cerberus provides implicit causal ordering by performing causal validations, which compare the global and local version vectors. The global and local version vectors capture the global and local version information known regarding the different data items related to an entity. Intuitively, for any operation accessing a particular data item in a transaction T, if the version entry corresponding to any representation in the local version vector of a data item is greater than the corresponding entry in the global version vector of the entity associated to the item, it means that a transaction causally ordered after T has modified or read the data item. In that case, the transaction is aborted. Otherwise, the causal validation passes successfully

Data Item	Value	Update Version Vector (UVV)	Read Version Vector (RVV)
x_1	1	[0 1 2]	[0 1 3]
y_1	2	[1 4 2]	[1 4 2]

Fig. 4: Metadata at each representation in Cerberus

and transaction commits. We now give a detailed description of each phase of a transaction in Cerberus.

Read Phase: The application client sends the read-set of the transaction to the OPL, comprising the data items to be read. When a transaction, T, arrives at the OPL, it reads the GVV of entity x corresponding to the transaction, denoted as $gvv(x,T)$. The GVV of an entity is only read once during a transaction. The OPL then sends the read requests to the corresponding representations, where the data items are stored. The request to a representation comprises the data item to be read and the GVV of the entity related to the corresponding data item. The read request is sent to the Cerberus layer at each representation, which then sends the request to the datastore layer of the representation. The datastore layer atomically reads the value as well as the metadata associated with the data item. The metadata comprises the UVV and the RVV associated with the data item, referred to as $uvv(x_i,T)$ and $rvv(x_i,T)$, the UVV and RVV of x_i for transaction T. The Cerberus layer then performs a check, defined as a *causal read check* to validate that the read data item, x_i, satisfies the causal dependencies related to entity x, across representations. The causal read check compares the GVV of the entity to the UVV of the data item to validate that the read operation follows the implicit causal order. For a read of data item x_i to be valid, the GVV of the corresponding entity, x, $gvv(x,T)$, should be greater than or equal to the UVV associated to the data item, $uvv(x_i,T)$. If the read does not pass the causal read check, then the transaction is aborted and the decision is sent to the OPL. Otherwise, the read operation is successful.

If the read is successful and the RVV of the data item is less than the GVV of the transaction, then the RVV of the data item, $rvv(x_i,T)$ is updated to the GVV, $gvv(x,T)$. The RVV is updated by sending a request to the datastore layer. The datastore layer uses an atomic test-and-set operation to update the data item. The test-and-set ensures that the RVV of a data item is updated only if the item is not concurrently updated after the read validation. This guarantees that the causal read check is still valid. If the read is successful, then the read result comprises the value of the data item read and the RVV and UVV of the data item. The OPL collects the read results from all representations and sends them to the application client. Note that RVV is only updated by a read when there has been an update to any representation associated with the entity, after the most recent read of the data item. Hence the protocol has minimal overhead for a read-heavy workload.

Write Phase: If the read phase is successful and the transaction is read-only, then the transaction is committed. Otherwise, in the second phase, the application client sends the write request to the OPL. The OPL then sends the write to the representation to be modified. As noted above, writes are performed on a single representation. Each write request contains the data item, x_i, the value to be written, $value_{x_i}$, and the GVV of the entity x read by the transaction, $gvv(x,T)$. If the transaction performed a read phase at the representation to which the write-request is being sent, then the request also contains the local version vectors, the UVV and the RVV of the data item read during the read-phase. If the data item, x_i was not present in the read set, then the Cerberus layer first sends a read request to the datastore layer. This action is performed to access the local version vectors, RVV and UVV corresponding to the data item x_i at the particular representation. The RVV and the UVV are needed to validate if the write operation follows implicit causal ordering. Similar to the read phase, the GVV, $gvv(x,T)$ is compared against the local version vectors (UVV and RVV) of x_i, to verify that the write operation follows the causal order across representations. This check is defined as a *causal write check*. Since implicit causality in Typhon implies that a write operation on an entity depends on preceding reads and writes, causal write check verifies whether the GVV is greater than or equal to both the respective UVV, as well as the RVV. If the RVV is greater, it means that a transaction causally ordered after the current transaction, has *read* the data item. On the other hand, if the UVV is greater, it means that a transaction causally ordered after the current transaction, has *written* the item. If the causal write check is successful, then the commit phase is processed for the transaction. If the check fails, the transaction is aborted.

Commit Phase: If the write obeys causal relationship across representations, the commit phase is performed, in which the datastore layer updates data item. The datastore layer uses an atomic test-and-set operation to update the value and the corresponding metadata related to the data item. The test-and-set verifies that the RVV and the UVV have not been modified after the causal read and write checks. If the RVV was updated during the read-phase by the current transaction, the test-and-set compares with the updated RVV. This mechanism ensures that the causality checks performed are still valid. Along with writing the new value of the data item, the UVV is also updated. The new UVV for the modified data item, x_i, is equal to the GVV of entity x, corresponding to the transaction, $gvv(x,T)$, except the entry for x_i. The i^{th} entry in $uvv(x_i)$, corresponding to x_i at R_i, is generated by monotonically incrementing the previous version number of x_i at R_i.

Post Commit Processing: The commit decision is then sent to the OPL. If the commit is successful, the new version number of the modified data item is sent back to the OPL, along with the commit decision. If x_i is the updated data item, the latest version of data item x_i is sent. The OPL then updates the GVV for the entity modified by the transaction to reflect the updated data item. After the GVV is modified, the commit decision is sent back to the application client. If the transaction is aborted, the OPL just returns the decision to the client, without performing any updates to the GVV.

Cerberus would not allow the inconsistent execution described in Section II. Causal read check for transaction T_3 would fail at the key-value representation, as the local version entry in the UVV of x_1 corresponding to the graph representation would be greater than the corresponding entry in T_3's GVV. T_3 would be then aborted. Cerberus employs a lightweight version-vector based approach, to provide implicit causal order for data access across representations.

IV. Scaling and Recovery

Scaling. Cerberus handles the large scale of the data by *scaling-out* via data partitioning. Both the data and the metadata at the OPL can be partitioned. Data at each representation can be partitioned among multiple servers. The OPL has information about the location of different data items at a particular representation and routes each read or write request accordingly to the appropriate representation server. Additionally, to handle a high volume of requests, Cerberus can scale-out by using multiple OPL servers. The logical entities are then partitioned among the multiple OPL servers. As each transaction only accesses a single entity, each transaction is processed at a single OPL server. The OPL and the representation servers can be scaled independently of each other. Cerberus's architecture and execution model enables it to seamlessly scale to handle the large-scale of the data.

Recovery. Cerberus can recover from crash failures of both representations as well as OPL servers. Each representation maintains and persists its own redo log. The datastore layer of a representation ensures that each committed operation is written to the redo log. Note that the updates to the RVV during reads are treated the same way as updates to the items, and are also logged. This redo log can then be replayed to recover each representation after a crash failure. A crash failure can occur after a write transaction has committed at a representation and before the latest version of the data item was sent to the OPL. Hence, on recovery, each representation sends the latest version of its data items to the OPL. On the failure of a representation, reads and writes to other representations can still proceed.

If the OPL fails, Cerberus becomes unavailable. If the metadata is partitioned among multiple OPL servers, then on the failure of an OPL server, the transactions on the entities maintained by the failed server cannot proceed.

To recover after crash failures, the OPL server(s) has to contact the appropriate data representations. Each representation then sends the latest version of all the data items to the OPL. Each OPL server uses this information to update the GVV of all the entities it maintains. OPL only maintains the version vector metadata in main-memory. This helps in reducing the commit latency but increases the time of recovery of the OPL. The failure of an OPL server only affects liveness, and does not affect correctness. The OPL can be made highly available by replication using Paxos [20] or other replication mechanisms.

V. Evaluation

A. Benchmark Description and Baselines Used

We develop a transactional micro-benchmark based on the YCSB benchmark [9] to issue operations accessing data across representations. Each transaction groups multiple single-key read and write operations on data items related to a single entity.

Read-only transactions read 2 data items related to an entity, at 2 different representations.

Read-write transactions read a data item and then write another data item related to the same entity. 50% of the read-write transactions have the read and write on different representations.

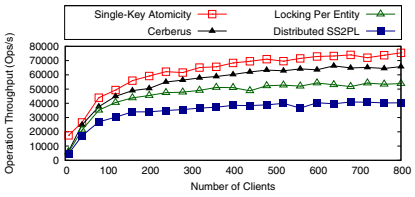

Fig. 5: Scaling-Up Clients - Uniform Distribution

Baselines. We compare Cerberus with three contrasting baselines, one providing *weak consistency* guarantees and the other two providing *strong consistency* guarantees on operations accessing data across representations.

Single-Key Atomicity scheme executes atomic read or write operations, and provides no consistency guarantees on operations accessing data across representations.

Distributed Strong Strict Two-Phase Locking (Distributed SS2PL) [6] is a commit-order preserving concurrency control protocol which ensures that the commit order between all the non-concurrent transactions is preserved. Hence, distributed SS2PL satisfies the consistency guarantees across representations defined in Typhon. Each transaction first acquires locks on all the items its accesses. All the locks (both read and write locks) are released at the end of the transaction.

Locking Per Entity scheme provides a serializable order of transactions. The Locking per entity scheme also satisfies the consistency guarantees defined in Typhon. An operation accessing a data item related to an entity leads to a lock on the entire entity. This orders all the transactions that access a particular entity. Locking the entity provides the advantage that if a transaction reads data from multiple representations, it only has to acquire a single lock.

B. Experimental Setup

The experiments were performed on a local cluster. Each machine in the cluster runs a 8-core Intel Xeon E5620 processor clocked at 2.40 GHz and has 32 GB of RAM. The experimental setup comprises application clients, OPL and data representations. Our baseline setup employs 12 machines in total: 8 as client machines, 1 as an OPL server and 3 as data representation servers.

The dataset comprises 1 million entities, and each entity has a data item related to it at each of the 3 corresponding representations. Unless otherwise mentioned, we employ our baseline setup and the workload is uniformly distributed over the entities, and comprises 80% read-only transactions and 20% read-write transactions.

At each representation, data is stored using an *in-memory key-value store*. We develop a key-value interface which supports get, put and test-and-set operations. Logging to the disk (Section IV) is turned off. Java is used for implementation, and Google's GRPC [2] and protocol buffers [3] for network communication. All the read and write requests are blocking calls. In every experiment, each client issues 10,000 transactions.

C. Experimental Results

1) Scaling Up Clients

We analyze Cerberus's performance by scaling the number of clients, as illustrated in Figure 5. The Single-key atomicity scheme achieves the highest throughput. With 800 clients, it achieves a throughput of 75.42K operations/sec and performs

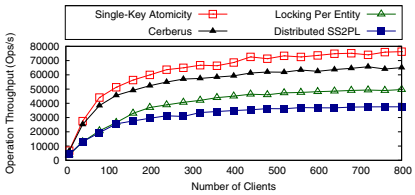

Fig. 6: Scaling-Up Clients - Zipfian Distribution

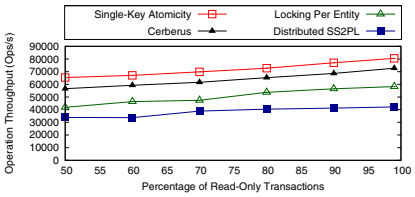

Fig. 7: Varying Read Write Percentage

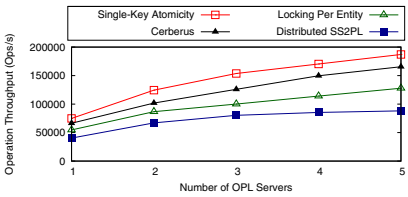

Fig. 8: Scaling OPL

15% better than Cerberus. Cerberus achieves an throughput of 65.75K operations/sec and performs 22% better than the locking per entity scheme and 63% better than Distributed SS2PL. Every operation in the Single-key atomicity scheme just atomically reads or writes the data item and no other processing is done to provide consistency guarantees. Hence, although Single-key atomicity performs well, it can lead to inconsistent semantic executions, like the one described in Section II. Locking schemes perform the worst, but provide the strongest consistency guarantees. Cerberus provides a trade-off in performance and consistency guarantees. Cerberus performs worse than the Single-key atomicity scheme and better than the locking schemes, and still provides implicit causal order guarantees for accessing data across representations.

Among the locking schemes, locking per entity scheme performs better than Distributed SS2PL. We attribute this to the fact that the locking per entity scheme has to acquire and release less number of locks per transaction, since it acquires the lock at the granularity of the entity. Less than 0.05% transactions were aborted for all the schemes.

2) Non Uniform Distribution

Next, we analyze the performance of Cerberus using a non-uniform Zipfian distribution, with zipfian constant (theta) set to .99 (Figure 6). The Single-key atomicity scheme performs the best and Cerberus performs better than the locking schemes. The performance gap between Cerberus and the locking schemes increases as compared to the case with uniform distribution. With 800 clients, Cerberus performs 31% better than the locking per entity scheme and 73% better than Distributed SS2PL. Cerberus provides implicit causal order, which only enforces an ordering among the non-concurrent transactions across the representations, at the granularity of a single entity. This allows Cerberus to better utilize the concurrency for operations accessing the same entity across representations, under higher contention. Furthermore, as Cerberus does not employ locking, it does not suffer from lock contention. These results show that even though Cerberus is optimistic, it performs well under contention and achieves a higher throughput than the pessimistic locking schemes.

For both Cerberus and locking schemes, we see more aborts as compared to the case with uniform distribution. 7.18% of the transactions are aborted for the locking per entity scheme, 5.4% for Distributed SS2PL and 3.17% for Cerberus. This is due to the contention on the entities accessed more frequently in the workload. Cerberus observes 3 types of aborts: *Causal read check violation* (amounts for 53% of the aborts), *Causal write check violation* (46.5%) and *Test-and-set failure* due to concurrent data access (0.5%). These numbers show that under contention, Cerberus's causal checks help in preventing a significant number of potential consistency violations.

3) Varying Read-Write Percentage

In Figure 7 we vary the percentage of read-only transactions from 50 to 99. The number of clients is fixed to 600.

As the percentage of read-only transactions increases, the throughput of all the four schemes increases, as the reads are less expensive than the writes in the in-memory store. The performance of Cerberus increases at a slightly higher rate. This is because as the percentage of writes decreases, the amount of updates to the RVV (read version vector) decrease as well. RVV is updated during a read of a data item, if there has been an update to another data item related to the same entity, after the recent read of the item. With 50% read-only transactions, around 38% of the reads update the RVV. When the read-only transactions go up-to 99%, then only .8% of the reads lead to updating the RVV. This makes Cerberus well suited to read-heavy workloads like social networks [24].

4) Scaling OPL

Figure 8 studies scaling of Cerberus. The number of OPL servers is varied from 1 to 5, with the number of representation servers being fixed at 3. The entities are range partitioned among the OPL servers equally. Each OPL server processes the transactions which access an entity in the range managed by the server. The number of clients is fixed to 600.

Both Single-key atomicity and Cerberus scale better than the locking schemes. With 5 OPL servers, Cerberus achieves a throughput of 165K operations/sec, which is 30% better than the locking per entity scheme and 88% better than the Distributed SS2PL. Scaling the OPL servers provides an advantage to all the schemes, as multiple OPL servers can harness more network bandwidth and processing, and are able to better utilize the cpu at each representation. In Cerberus, OPL only needs to read and update the GVV for the entity accessed by a transaction. Hence, it is suited to scaling OPL servers, as the reading and updating the GVV is now partitioned among multiple servers. The locking schemes still have to acquire locks during each transaction, which restricts the scaling in throughput, as compared to Cerberus. These results show that Cerberus can efficiently scale via the use of multiple OPL servers.

VI. Related Work

Our work builds on and is related to prior work in the areas of heterogeneous databases, causal consistency and various consistency models in databases.

Heterogeneous Databases. Polystores like BigDAWG [13] aim at managing data in heterogeneous engines and provide a query layer over multiple DB engines. However, the updates are local to each data source and the system does not provide any consistency semantics for updates to multiple data sources. Sagas [17] execute transactions over heterogeneous databases by dividing transactions into sub-transactions, and executing the sub-transactions in a pre-defined order. Although Sagas allow efficient execution of long-lived transactions, it provides weaker guarantees than Cerberus, and can lead to the inconsistent executions discussed earlier. Dey et al. [12] provide snapshot isolation guarantees for transactions executing across heterogeneous key-value stores. Deuteronomy [21] separates transactions from the data component and can support ACID transactions over hybrid data stores. Cerberus provides implicit causal order across heterogeneous data, which is weaker than global serializability but is less expensive to enforce.

Breitbart et al. [7] developed the notion of strong serializability to define serializable schedules, which preserve the order of non-concurrent transactions. Typhon's implicit causal dependencies also preserve the order between non-concurrent transactions, but only for transactions accessing the same logical entity, which leads to more permissible histories. Other weaker correctness criterion such as local serializable (LSR) [16], 2LSR [26] have also been proposed.

Causal consistency. Lamport [19] introduced the concept of happens-before relationship between events in a distributed system. Many systems provide causal consistency [22], [23] over replicated data. COPS [22] and Eiger [23] provide convergent causal consistency in a geo-replicated setting using explicit dependencies. Each replica only commits an operation after the explicit dependencies have been satisfied. Typhon builds on the notion of potential causality [19] and uses vector clocks [15], [25] to satisfy implicit dependencies at the granularity of an entity in a heterogeneous setting, rather than using explicit dependencies. Cerberus uses both read and update version vectors, due to which it can support read-write transactions, apart from read-only and write-only transactions supported in COPS and Eiger.

Others. Spanner [10] provides external consistency using True-time API in a wide-area setting, ensuring a global order on all operations. However, external consistency needs either atomic clocks, which are expensive or some other method of time synchronization [18]. Deterministic schemes [29], [14] can also be used to provide a global order on all operations. A locking solution like Distributed SS2PL [6] built over all the items at all the representations, preserves the commit order between all non-concurrent transactions. Cerberus enforces implicit causal order on operations at a fine granularity of a single entity and does not need a central sequencer like deterministic schemes or locking managers like in SS2PL.

Megastore [5] (transactions access a fixed key-group) and G-Store [11] (transactions access dynamically created key-groups) also restrict the scope of transactions, similar to the single-entity transaction model in Cerberus.

VII. Conclusion

We propose Typhon, which defines a consistency model for accessing application data stored across multiple representations. Typhon uses entities to link data across different representations. Typhon introduces new implicit causal edges in the conflict graph model to capture logical dependencies among operations accessing data across representations.

We also present a protocol Cerberus, which provides implicit causal guarantees introduced in Typhon. Evaluation results show that the performance of Cerberus is close to a solution providing no consistency guarantees across representations and better than locking based solutions.

Acknowledgements This work is supported by NSF grant CSR 1703560.

References

[1] Couchbase. http://www.couchbase.com/.
[2] GRPC. http://www.grpc.io/.
[3] Protocol Buffers. https://developers.google.com/protocol-buffers/.
[4] Voldemort. http://www.project-voldemort.com/.
[5] J. Baker, C. Bond, J. C. Corbett, et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.
[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.
[7] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 23–56, 1992.
[8] N. Bronson, Z. Amsden, G. Cabrera, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, 2013.
[9] B. F. Cooper, A. Silberstein, E. Tam, et al. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*, pages 143–154, 2010.
[10] J. C. Corbett, J. Dean, M. Epstein, et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):8, 2013.
[11] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SOCC*. ACM, 2010.
[12] A. Dey, A. Fekete, and U. Röhm. Scalable distributed transactions across heterogeneous stores. In *ICDE*, pages 125–136, 2014.
[13] A. Elmore, J. Duggan, M. Stonebraker, et al. A demonstration of the bigdawg polystore system. *VLDB*, 8(12):1908–1911, 2015.
[14] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *VLDB*, 8(11):1190–1201, 2015.
[15] C. J. Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987.
[16] H. Garcia-Molina and B. Kogan. Achieving high availability in distributed databases. *IEEE Transactions on Software Engineering*, 14(7):886–896, 1988.
[17] H. Garcia-Molina and K. Salem. *Sagas*, volume 16. ACM, 1987.
[18] S. S. Kulkarni, M. Demirbas, D. Madappa, et al. Logical physical clocks. In *OPODIS*, pages 17–32, 2014.
[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
[20] L. Lamport. The part-time parliament. *ACM TOCS*, 1998.
[21] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, 2011.
[22] 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 *SOSP*, pages 401–416, 2011.
[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *USENIX NSDI*, pages 313–328, 2013.
[24] H. Lu, K. Veeraraghavan, P. Ajoux, et al. Existential consistency: measuring and understanding consistency at facebook. In *SOSP*, 2015.
[25] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.
[26] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *IEEE Parallel and Distributed Information Systems*, 1991.
[27] J. Shute, R. Vingralek, B. Samwel, et al. F1: A distributed sql database that scales. *VLDB*, 6(11):1068–1079, 2013.
[28] M. Stonebraker and U. Cetintemel. " one size fits all": an idea whose time has come and gone. In *ICDE*, pages 2–11, 2005.
[29] A. Thomson, T. Diamond, S.-C. Weng, et al. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.