Dejima: Decentralized Transactional Data Integration with Bidirectional Update Propagation

Yasuhito Asano[1], Yang Cao[2], Dennis-Florian Herr[3], Soichiro Hidaka[4], Zhenjiang Hu[5], Yasunori Ishihara[6], Hiroyuki Kato[7], Hsiang-Shang Ko[8], Kota Miyake[9], Keisuke Nakano[10], Makoto Onizuka[9], Yuya Sasaki[9], Toshiyuki Shimizu[11], Masato Takeichi[12], Van-Dang Tran[7], Kanae Tsushima[7], Yusuke Wakuta[9], Chuan Xiao[9], Masatoshi Yoshikawa[2]

[1]Toyo University, [2]Kyoto University, [3]Bosch, [4]Hosei University, [5]Peking University, [6]Nanzan University, [7]National Institute of Informatics, [8]Institute of Information Science, Academia Sinica, [9]Osaka University, [10]Tohoku University, [11]Kyushu University, [12]The University of Tokyo

yasuhito.asano@iniad.org,hidaka@hosei.ac.jp,huzj@pku.edu.cn,yasunori.ishihara@nanzan-u.ac.jp,kato@nii.ac.jp,ksk@riec.tohoku.ac.jp,onizuka@ist.osaka-u.ac.jp,yoshikawa@i.kyoto-u.ac.jp

ABSTRACT

In data integration, one of the essential tasks is to federate different systems to provide useful services. Despite a large amount of research on this topic, several issues are not yet fully addressed, such as read-only views, inflexibility, centralized design, and lack of consistency. Aiming to address these technical issues, we propose Dejima, a transactional data integration architecture. Designed in a *decentralized* fashion, Dejima allows each participant to autonomously manage its own database and collaborate with other participants. Multiple participants constitute a Dejima group and exchange data through Dejima tables specified by a global schema, thereby achieving *flexible* data integration. To propagate the updates in databases cascadingly in the network, Dejima provides participants with an interface and an efficient internal mechanism to support *updatable views* by employing bidirectional transformations. Dejima supports distributed transaction management to ensure *global consistency* for update propagation. To show the usability of the Dejima architecture, we discuss a case study for a ride-sharing alliance application. We implement a prototype of the Dejima architecture on top of PostgreSQL and evaluate its transaction management performance using the YCSB and TPC-C benchmarks modified for decentralized data integration.

PVLDB Reference Format:
Yasuhito Asano[1], Yang Cao[2], Dennis-Florian Herr[3], Soichiro Hidaka[4], Zhenjiang Hu[5], Yasunori Ishihara[6], Hiroyuki Kato[7], Hsiang-Shang Ko[8], Kota Miyake[9], Keisuke Nakano[10], Makoto Onizuka[9], Yuya Sasaki[9], Toshiyuki Shimizu[11], Masato Takeichi[12], Van-Dang Tran[7], Kanae Tsushima[7], Yusuke Wakuta[9], Chuan Xiao[9], Masatoshi Yoshikawa[2]. Dejima: Decentralized Transactional Data Integration with Bidirectional Update Propagation. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Information/Data silos are typical phenomena in large companies and organizations. A common practice is the collaboration or acquisition of companies: there is large demand for different systems to be federated to provide useful services to users, yet each company has its own goal and often builds its own applications and database systems independently without federating with others. In consequence, for system federation, we need to integrate the independently-built databases.

1.1 Issues in Prior-Art

Albeit a large amount of related research has been conducted for data integration (as summarized in the literature [29]), the following issues are not yet fully addressed.

- **Read-only views.** The first major issues of the current data integration systems is that "views are not updatable". In data integration systems, data is often exchanged between different databases through views. From a technical point of views, most of research focuses on read-only data exchange, since view update is a difficult problem caused by the ambiguity to source tables [50]: when a view is updated, there are potentially many strategies to translate the view update to an update to the base tables, so it is difficult to choose a suitable one automatically [50]. Most existing DBMSs provide limited support for view update, whereas an ideal data integration system should support updatable views: not only data sharing through views but also update propagation from views to source databases.
- **Inflexible data integration**. The second issue of the current data integration systems is their inflexibility. They are based on either assuming a single shared global schema [35, 73] or a peer-based approach without a shared schema [49, 51, 63]. The former is mainly designed for enterprise use cases, where data is exchanged between a small number of databases through a single global schema. The latter is designed for the cases where there are large number of databases so it is difficult to decide a single global schema;

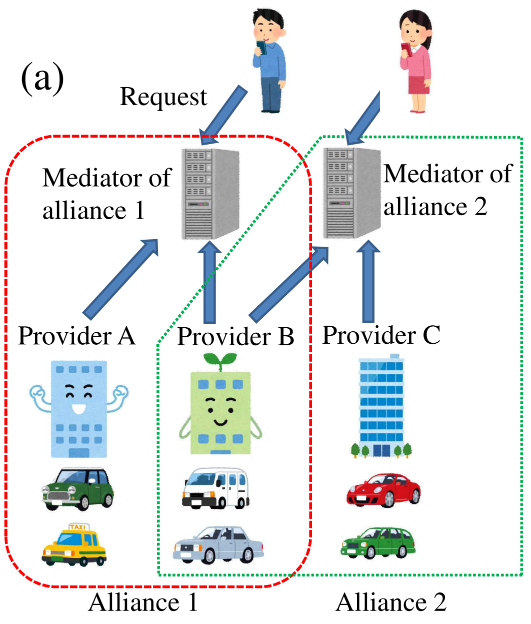
instead, data is exchanged between peers (each manages its own database) and the update propagates cascadingly in the peer network. However, the problem of data integration is not simple enough to fall into the above two categories. An ideal data integration system should consider both aspects and combine both advantages of the two approaches.

- **Centralized design**. Many existing systems were designed in a centralized fashion, while from the privacy-preserving perspective, it is recommended that the systems to be federated are able to manage services without a centralized (third-party) mediator, which has to be trusted by all the peers. Hence a decentralized design is preferred.
- **Lack of global consistency**. In the current peer-based data integration systems, each peer is able to update its own database, and local consistency at each peer is ensured [42, 49]. However, the update requests may contradict each other when integrated, and may violate the fine-grained access control and auditing requirements for collaborations and sharing [4], rendering existing techniques inapplicable for data integration tasks that demand global consistency.

To show a motivating example for the above issues, consider a target application: a ride-sharing alliance system. It allows non-professional drivers to provide taxi service using their vehicles. Each driver/vehicle belongs to a single ride-sharing company (e.g., Uber, Didi, etc.). As the size of the ride-sharing market increases, a number of ride-sharing service providers have made a partnership with other providers to increase the chance of matching providers to customers [70]. For example, Uber and Didi have merged in China business, and Ola Cabs and GrabTaxi are in talks for a global taxi alliance. We call such a set of providers a *ride-sharing alliance*. Figure 1 illustrates an example of ride-sharing alliances: providers *A* and *B* belong to Alliance 1 and providers *B* and *C* belong to Alliance 2.

To integrate data and provide service for a ride-sharing alliance, a conventional approach employs a *mediator*, a single representative provider or a trusted third party to manage the database of vehicles available on the alliance, as illustrated in Figure 1 (a). Each provider sends the data of every vehicle that it has decided to make available in the alliance to the mediator. When a customer (i.e., passenger) books a taxi, the mediator receives the request and matches it to a proper vehicle. The request is then sent to the provider, which updates its local database to complete the booking.

The above approach is popular in various kinds of alliances, including Amazon marketplace that can be considered as an alliance of sellers. This approach, however, has the following drawbacks from a viewpoint of data integration: (1) The mediator is likely to have excessive authority, because it is difficult to find a trusted third party who does not want profit. For example, in 2019, Rakuten, a mediator of a seller alliance in Japan, forced sellers to shoulder shipping costs. (2) Each provider has to write a program to manage its local database to select vehicles available in the alliance and update it according to the information sent from the mediator. If a provider takes part in multiple alliances, such a program could be more complicated because it requires global consistency. For example, the provider *B* that belongs to the two alliances might receive two contradicting requests from the mediators of them at



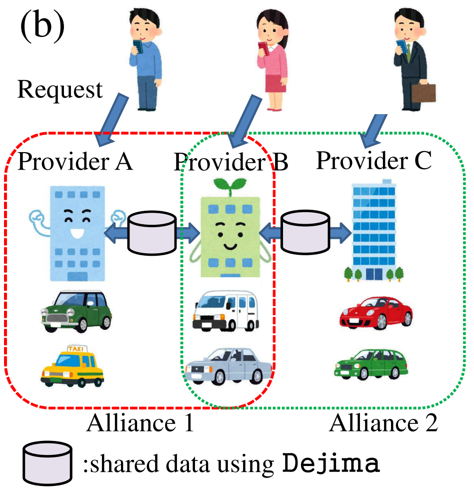


Figure 1: (a) Ride-sharing alliance system by conventional data management, and (b) Ride-sharing alliance system by Dejima.

the same time (e.g., two alliances try to assign the same vehicle to different customers simultaneously). A system that guarantees only local consistency is unable to address the issue. On the other hand, when views are read-only in the database, it is difficult for a mediator to prepare a program for sharing data between its global database and the providers' local databases, because the providers either have to assume a global schema that endows the mediator with more authority, or independently design local schemas, which are eventually hard to integrate.

1.2 Our Approach

To the above research challenges, we present Dejima[1], a *decentralized* transactional data management architecture that supports *updatable views* and integrates databases in a *flexible* manner [8, 9, 41]. The Dejima architecture is influenced by ORCHESTRA [42, 49]. It shares the same idea of the peer-based data exchange and newly introduces update strategy for avoiding view update ambiguities. Furthermore, the Dejima architecture introduces the notion of Dejima group for effective data exchange among multiple participants [2] and puts emphasis on supporting *global consistency*, whereas ORCHESTRA handles consistency only locally inside of each participant. The summary of our contributions in the Dejima architecture is described as follows:

- **Bidirectional view update**. Dejima provides a simple interface for users and an efficient internal mechanism to support updatable views. By employing bidirectional transformations [9, 72], the architecture enables users to easily write update strategy for avoiding the ambiguity of update propagation from views to source databases. The update is conducted efficiently by incremental view maintenance and view update.
- **Flexible database integration.** Dejima combines the advantages of both the global schema-based approach and the peer-based approach: multiple participants form a *Dejima group* and they can exchange data by using *Dejima tables*, which are updatable views specified by a global schema locally in the group. In addition, if a participant belongs to multiple Dejima groups and their Dejima tables are not independent[3], update propagates between multiple Dejima tables in a similar way as does peer-based data integration [49].
- **Decentralization.** Dejima is designed in a decentralized fashion such that each participant can update its own database and the update is propagated to other participants via Dejima tables. The data available on each participant is integrated and managed without resorting to third-party mediators nor writing in a complicated program for consistency.
- **Global consistency via distributed transaction management.** Moreover, once we permit updatable views and decentrailized data integration, update may cascadingly propagate to multiple participants. To achieve consistent update propagation, Dejima supports distributed transactions. The major difficulty of supporting global consistency in data integration systems is that the scope of the update propagation is identified only at execution time. This causes a higher rate of update conflicts as update propagates more participants. Dejima solves this issue by managing update propagation scope in record-level and participant-level that guarantees global consistency and achieves high throughput.

We apply the Dejima architecture on the aforementioned example of ride-sharing alliances. As shown in the decentralized data integration illustrated in Figure 1 (b), each customer sends a request to a favorite provider, and the provider matches it to a vehicle available in the alliances to which the provider belongs. In the Dejima architecture, each participant represents a provider and each Dejima group represents an alliance. Distributed transactions are managed for taxi booking: When a customer books a taxi, the provider attempts to update its own base table (only viewable to the provider itself), and then the update is propagated to Dejima tables (each viewable to the providers in an alliance) and cascadingly propagated to other participants' base tables through the updatable view mechanism, thereby addressing the potential conflicts, i.e., requesting other alliances not to assign the vehicle to other customers simultaneously.

We implement a prototye of the Dejima architecture on top of a DBMS (PostgreSQL) and evaluate its transaction management performance by running it on DBMS benchmarks (YCSB and TPC-C) modified for a decentralized data integration scenario. We compare the proposed solution to a strict two-phase locking method, as well as a hybrid method which enables each participant to autonomously select between our protocol and two-phase locking for better throughput. The experimental results indicate that our method consistently achieves better throughput (up to 1.4x) on workloads with high contention, while users are suggested to use two-phase locking or the hybrid method for workloads with low contention.

1.3 Paper Outline

The rest of the paper is organized as follows. Section 2 introduces the overview of the Dejima architecture for flexible and decentralized data integration. Section 3 presents the bidirectional view update. The distributed transaction management for global consistency is covered by Section 4. Section 5 briefly presents the system implementation of Dejima architecture. Section 6 discusses the use of Dejima in a ride-sharing alliance application. Experimental evaluation is reported in Section 7. Section 8 reviews related work. Section 9 concludes the paper.

2 DEJIMA ARCHITECTURE

In this section, we describe an overview of the Dejima architecture and its mechanism of update propagation by employing bidirectional transformation (BX).

2.1 Software Components

The Dejima architecture consists of two components, participants and Dejima groups [8]. Each participant manages its own local database and participates in Dejima groups for sharing data with other participants. Client applications submit transactions to local databases (typically relational databases) at each participant. The Dejima architecture architecture manages distributed transactions by view update propagation. Specifically, participants exchange data/propagate update to other participants through Dejima tables in a Dejima group. Dejima tables are updatable views defined by using BX languages [9] so that we achieve update propagation between base tables (source data) and Dejima tables (target data). Dejima tables can be seen as a global schema locally in each Dejima group.

[1]Dejima was a small, artificial island located in Nagasaki, Japan. All the trades between Japan and foreign countries were made through Dejima from 1641 to 1854. We name our architecture for the resemblance to Dejima in functionality.

[2]To avoid ambiguity, we use the term "participant" rather than "peer" in this paper, since our solution differs from peer-based approaches.

[3]That is, an update of one Dejima table causes an update to the source base tables, and then the latter update causes an update to another Dejima table.

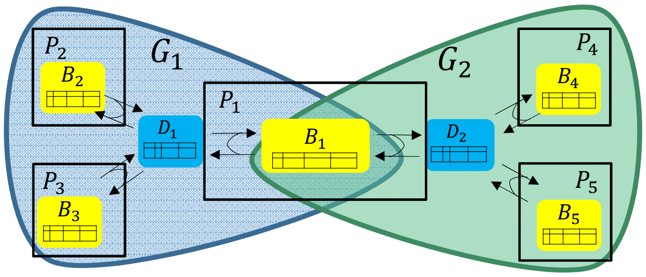


Figure 2: An example of Dejima architecture: P_n, G_n, D_n, B_n **indicate participant, Dejima group, Dejima table, and base table, respectively.**

Figure 2 shows an example of the Dejima architecture which consists of five participants $P_1, \ldots, P_5$ and two Dejima groups G_1 and G_2. Each participant P_i has its own base table B_i and participates in the Dejima groups. Dejima table D_1 is shared by P_1, P_2, P_3 and D_2 is shared by P_1, P_4, P_5. Conceptually, Dejima table is replicated among the participants in the same Dejima group.

2.2 Update Propagation

Once a base table is updated by a client application, the update is cascadingly propagated to other participants through BX. The procedure works as follows.

(1) An update made on a base table is propagated to a Dejima table through the BX defined between the base table and the Dejima table.
(2) The update made on the Dejima table is synchronized to the other participants in the same Dejima group.
(3) The synchronized update on the Dejima table is propagated to the base table at each participant [4] through BX.
(4) We may have multiple Dejima groups for a base table. In such case, an update may propagate between those Dejima groups and then continue the cascading update propagation to other participants.

For example, if we receive an update on B_2, it is transformed to the update on D_1 and propagated to D_1. The update made on D_1 in P_2 is synchronized with D_1 in P_1 and P_3. Then, the synchronized update made on D_1 is propagated to B_1 in P_1 and B_3 in P_3. The update made on B_1 may propagate to D_2 depending on the definition of the BX between B_1 and D_2.

Since the Dejima architecture features a global schema, its design resembles more a mediator-wrapper style than a peer-based style. In this sense, Dejima targets application scenarios with tens (up to 100) of participants, and we assume the network is more stable (i.e., adding/deleting a participant is infrequent) than the scenario targeted by peer-based solutions. If a participant joins the network, the global schema needs modification to reflect its role in the network; e.g., by sharing a Dejima table in a Dejima group. In this case, any uncommitted transaction that involves this Dejima group is aborted, the new participant's copy of the Dejima table is synchronized, and then update is made on its base table by BX. If a participant leaves the network, uncommitted transactions that involve this Dejima group are aborted, and then the global schema, along with any application logic that involves the leaving participant, is modified to reflect the removal of the participant.

In the following two sections, we present the details of the Dejima architecture from two aspects. The first is the intra-participant aspect: how we achieve update propagation between base tables and Dejima tables bidirectionally. This mainly discusses database theory and programming language issues. The second is the inter-participant aspect: how we manage transactions for global consistency in the network. This mainly addresses the issues of transaction processing and concurrency control in a distributed environment. We also note that the focus of this paper is view update and transaction management, and we leave other issues such as crash recovery to future work.

3 BIDIRECTIONAL VIEW UPDATE

In this section, first we briefly present our approach proposed in [72] to the view update problem in relational databases. The key idea is to allow users to directly describe their intended view update strategies, thereby solving the inherent ambiguity of view updating. Specifically, we use a class of the Datalog language as a formal language of view update strategies that the user-written strategies can be validated and optimized to run correctly and efficiently in commercial RDBMSs. Then we discuss how to use the approach in the Dejima architecture.

3.1 Bidirectional Transformations

Consider a view V defined by a query *get* over a source database S, we allow updates on the view by propagating them to the source. This update propagation can be formulated as a so-called *putback transformation put* [20, 31], which takes as input the original source S and an updated view V' to produce a new source S' as follows:

$$\begin{tikzcd} S \arrow[r, "get"] \arrow[d, bend left, start anchor=east, end anchor=east] & V \arrow[d] \\ S' & V' \arrow[l, "put"] \end{tikzcd}$$

The pair of *get* and *put* forms a BX [20]. To ensure consistency between the source database and the view, a BX must be well-behaved in the sense that it satisfies the following *round-tripping* properties:

$$\forall\, S, \qquad put\,(S,\ get(S)) = S \qquad \text{(GetPut)}$$
$$\forall\, S,\ V', \qquad get\,(put\,(S,\ V')) = V' \qquad \text{(PutGet)}$$

The GetPut property ensures that unchanged views correspond to unchanged sources, while the PutGet property ensures that all view updates are completely reflected to the source such that the updated view can be computed again from the query *get* over the updated source.

Clearly, a putback transformation captures an update strategy for propagating updates from the view to the source. More interestingly, while there may be many putback transformations for a view definition *get*, there is at most one view definition for a putback transformation *put* to form a well-behaved BX [30, 52]. We say a

[4]Since we assume that getput and putget laws are valid for BX [9], we don't need to make update propagation back to the source base tables.

putback transformation is valid if its corresponding view definition exists.

The uniqueness of view definition makes putback transformation essential for BX. Although *put* is written in a unidirectional (putback) manner, if *put* is valid, it uniquely determines the view definition *get* and thereby fully captures a well-behaved BX. In the next subsection, we shall introduce how putback transformations, i.e., view update strategies, can be programmed and validated.

3.2 View Update Strategies in Datalog

A putback transformation (i.e., view update strategy) can be completely specified by a query *putdelta* that computes source updates over the original source and the updated view. Indeed, we obtain the new source by applying the result of *putdelta* to the original one. We now show that *putdelta* can be written in the Datalog language [16] by using delta relations.

For each relation, an arbitrary update can be represented by a set of tuples inserted/deleted into/from the relation [53]. We use the concept of delta relations [33] to capture both these insertions and deletions. In this way, *putdelta* can be written as a Datalog program consists of many Datalog rules defining delta relations from the original source relations and the updated view. The following example illustrates a *putdelta* program.

Example 1. *Consider a source database S, which consists of two base binary relations, $r_1(X,Y)$ and $r_2(X,Y)$, and a view relation v defined by a union over r_1 and r_2. The following is a Datalog program that describes an update strategy for the view.*

$$\begin{aligned} -r_1(X,Y) &\text{ :- } r_1(X,Y), \neg v(X,Y). \\ -r_2(X,Y) &\text{ :- } r_2(X,Y), \neg v(X,Y). \\ +r_1(X,Y) &\text{ :- } v(X,Y), \neg r_1(X,Y), \neg r_2(X,Y). \end{aligned}$$

Here, we use "+" and "−" preceding a relation symbol to denote insertion and deletion operations on the relation, respectively. The first two rules state that if a tuple $\langle X,Y\rangle$ is in r_1 or r_2 but not in v, it will be deleted from r_1 or r_2, respectively. The last rule states that if a tuple $\langle X,Y\rangle$ is in v but in neither r_1 nor r_2, it will be inserted into r_1. Consider an original source instance $S = \{r_1(1,2), r_2(2,3), r_2(4,5)\}$ and an updated view $V = \{v(1,2), v(3,4), v(4,5)\}$[5]*, the result of putdelta is $\Delta S = \{+r_1(3,4), -r_2(2,3)\}$ that means tuple $\langle 3,4\rangle$ is inserted into r_1 and tuple $\langle 2,3\rangle$ is deleted from r_2. By applying ΔS to S, we obtain a new source database $S' = \{r_1(1,2), r_1(3,4), r_2(4,5)\}$.*

We extend Datalog rules by allowing a truth constant *false* (denoted as $\perp$) in the rule head for expressing integrity constraints. For example, consider a relation $r(X,Y)$, to prevent any tuples having $Y > 1$ in r, we can use the following constraint:

$$\perp \text{ :- } r(X,Y), Y > 1.$$

That means there is no tuple $\langle X,Y\rangle$ satisfying $Y > 1$ in r.

Since *putdelta* is a hand-written Datalog program, it must satisfy certain properties to guarantee every view update is correctly propagated to the source. Firstly, a view update strategy should be well-defined in the sense that there is no ambiguity in updating the source. Therefore, in the output of *putdelta*, there must be no insertion and deletion of the same tuple on the same relation. Secondly, as presented previously, the putback transformation *put* specified by *putdelta* must be valid to determine a well-behaved BX between the source and the view. The putback transformation should also satisfy the round-tripping properties with the view definition expected beforehand.

The Datalog language for writing *putdelta* is expected to not only be expressive enough to cover many view update strategies but also has many good properties for deciding the correctness of the *putdelta* program. Here, we consider LVGN-Datalog [72] that is an extension of non-recursive Datalog with guarded negation, built-in predicates, constant and linear view predicate. LVGN-Datalog inherits the decidability of query satisfiability in guarded negation Datalog [13]. Our validation algorithm statically checks the correctness of the *putdelta* program by reducing it to the decidable query satisfiability problem. The algorithm is both sound and complete for any view update strategies written in LVGN-Datalog.

We further optimize the *putdelta* program by exploiting its well-behavedness and integrating it with the standard incrementalization methods for Datalog. It is remarkable that due to the GetPut property, over the original source and original view, *putdelta* results in no update on the source. An update on the view leads to some changes in the output of *putdelta* and therefore makes the source updated. This is the key observation to optimize *putdelta* by transforming it into an incremental program $putdelta^{inc}$ (also in non-recursive Datalog) that compute source updates from view updates more efficiently. Since *putdelta* is a Datalog program, our incrementalization algorithm adopts the incremental view maintenance technique for Datalog introduced in [33].

The Datalog rules of delta relations in $putdelta^{inc}$ (or in *putdelta*) make the source updated whenever there are any view updates. To implement this reactive behaviour of $putdelta^{inc}$ in RDBMSs, we use the trigger mechanism [64]. Specifically, we compile $putdelta^{inc}$ into a SQL program that defines certain triggers and associated trigger procedures on the view. These trigger procedures are automatically invoked in response to view update requests, which can be any SQL statements of `INSERT`/`DELETE`/`UPDATE`. The triggers perform the following steps:

(1) Handling update requests to the view to derive the corresponding delta relation of the view.
(2) Checking the constraints if applying the delta relation from step (1) to the view.
(3) Computing delta relations of the source and applying them.

In our compilation, since there is no recursion in $putdelta^{inc}$, we translate Datalog rules defining delta relations into equivalent SQL queries which are called by the trigger procedures in step (3).

4 TRANSACTION MANAGEMENT FOR UPDATE PROPAGATION

We described a mechanism of update propagation in Dejima architecture in Section 2. Now, we have an important question: how can we ensure the global consistency and achieve high throughput when updates cascadingly propagate to multiple participants? A simple approach is to use a distributed version of strict 2PL, that is, we acquire locks on the way during update propagation in Dejima network. This approach is efficient when the contention rate of transactions is low. However, it suffers from deadlock, which

[5] We write $r(a,b)$ to denote a tuple $\langle a,b\rangle$ in r.

causes distributed transaction abort. We propose a new approach corresponding to a distributed version of conservative 2PL in order to avoid distributed deadlock. It acquires necessary locks before executing update propagation by appropriately designing update strategies using BX. This approach is efficient in particular when the contention rate is high. In detail, we first introduce two notions of expressing update propagation scope, family record set and update participant scope. The family record set expresses an atomic unit (distributed records) for distributed locking during update propagation; if two transactions access different family record sets, we ensure they do not conflict each other. The update participant scope effectively reduces the search space of participants for acquiring locks in conservative 2PL. Second, in order to ensure that the family record set is used as a distributed atomic unit in conservative 2PL, we design put rules for views defined with monotonic SPJU queries; the expressive power of the put rules is strictly less than LVGN-Datalog [13]. These rules ensure that BX always propagates updates on derived records back only to its original source records. Finally, we propose a hybrid mechanism that appropriately chooses either the simple approach (2PL mode) or the family record set based approach (conservative 2PL mode) according to the contention rate of transactions.

4.1 Family Record Set

We design family record set to express atomic unit for distributed locking during update propagation via BX. A family record set corresponds to a single virtual record (atomic unit) in a single virtual table in Dejima network. In detail, we construct a family record set when a new original record is inserted into a base table at some participant. This insertion is cascadingly propagated to other participants via BX and derives new records. We refer to the set of the original record and its derived records as a family record set. Family record set is implemented using lineage; we assign the same identifier for the records in the same family set. A benefit of using family record set is whenever a record in a family record set is updated, other records in the same family record set may be updated during update propagation, but other records not in the same family record set will never be updated. See Lemmas 1 and 2 in Section 4.3 for more detail.

Thus, we can achieve the global consistency and high throughput by avoiding distributed deadlock using family record set; we acquire write lock on the family record set of updating record before propagating updates in the Dejima network (conservative distributed 2PL), and then, we propagate updates using tree two-phase commit through a tree structure in the Dejima network.

4.2 Update Participant Scope

It may not be efficient to acquire locks using family record set in Dejima network, in particular when the number of participants is large. We introduce update participant scope, a set of participants to which an update may propagate via BX. The update participant scope effectively reduces the search space of participants for acquiring locks.

Consider a participant P participating in two Dejima groups G_i and G_j. Let get_i and put_i be get and put functions at P for Dejima group G_i. Define get_j and put_j similarly.

Let B be a set of base tables of P, and ud_i be an update operation on $get_i(B)$. We say that G_i *dynamically affects* G_j at P by B and ud_i if

$$get_j(B) \neq get_j(put_i(B, ud_i(get_i(B)))),$$

that is, update ud_i on the Dejima tables shared in Dejima group G_i causes some change on the one shared in Dejima group G_j.

Consider a family $\mathcal{B}$ of the sets of possible base tables of P, and a set $\mathcal{U}_i$ of "atomic updates" on possible Dejima tables of G_i. Typically, an atomic update is an insertion/deletion of one tuple to/from a Dejima table. We say that G_i *(statically) affects* G_j at P if there are some $B \in \mathcal{B}$ and $ud_i \in \mathcal{U}_i$ by which G_i dynamically affects G_j at P.

Now we can define *update propagation graph* as follows. The vertices are Dejima groups. A directed edge goes from one Dejima group G_1 to another group G_2 if and only if G_1 statically affects G_2 at some participant P. Using update propagation graph, update scope detection is straightforward. That is, the scope of an update issued by a participant P is all the participants participating a Dejima group which is reachable from some Dejima group that P participates in.

In order for this approach to work, the affecting relation must be decidable. As explained in Section 3.2, we have adopted LVGN-Datalog [72] as a query language for the BX between a base table and a Dejima table. If a query is written in LVGN-Datalog, the satisfiability of the query is decidable. Using this property, we can decide the affecting relation as follows: Construct two queries q_1 and q_2 in LVGN-Datalog such that

- q_1 returns all the tuples in $get_j(B)$ but *not* in $get_j(put_i(B, ud_i(get_i(B))))$, and
- q_2 returns all the tuples *not* in $get_j(B)$ but in $get_j(put_i(B, ud_i(get_i(B))))$.

G_i affects G_j at P if and only if at least one of q_1 and q_2 is satisfiable.

4.3 Put rules for SPJU queries

In order to ensure that the family record set is used as a distributed atomic unit in conservative 2PL, we introduce put rules for views defined with LVGN-Datalog for monotonic SPJU queries. First, for given view update Δv, we introduce a general put rule so that *put* is distributive over set union ($\cup$) and difference ($-$) by $\oplus$ and $\ominus$, respectively.

Definition 1 (Distributive put rule).

$$put(S, V \cup \Delta v) = put(S, V) \oplus put(S, \Delta v)$$
$$put(S, V - \Delta v) = put(S, V) \ominus put(S, \Delta v)$$

where $\oplus$ is a merge operator on source for old version ($put(S, V)$) and new delta ($put(S, \Delta v)$). If we assume bag semantics without primary key constraints on source, we simply utilize $\cup$ for $\oplus$ and $-$ for $\ominus$; the same record is identified between sets using lineage for $-$ operation. Otherwise (set semantics or bag semantics with primary key constraints on source), we abort the update when duplicate occurs.

Next, we define $put(S, \Delta v)$ for SPJU queries as follows to ensure the distributive property for relational operations:

Definition 2 (Put rules for SPJU queries).

$$\begin{aligned} selection &: \Delta s = \sigma_p^{-1}(\Delta v) = \sigma_p(\Delta v) \\ projection &: \Delta s = \pi^{-1}(\Delta v) \\ join(S \bowtie T) &: \Delta s = uniq_{S.id}(\pi_{S.*}(\Delta v)), \Delta t = uniq_{T.id}(\pi_{T.*}(\Delta v)) \\ union(S \cup T) &: \Delta s = \Delta v \ or \ \Delta t = \Delta v \end{aligned}$$

where $uniq_{atr}(S)$ is an operation that groups S using atr as a key attribute, and id is the primary key attribute. As an example for projection operation π, since $\pi(S \cup \Delta s) = \pi(S) \cup \pi(\Delta s) \rightarrow \Delta v = \pi(\Delta s)$, $put(S, \Delta v) = \Delta s = \pi^{-1}(\Delta v)$ holds[6]. Similarly, we design the above put rules for selection/join/union operations. As for the union operation, if Δv is originally derived from Δs or Δt, we identify Δs or Δt using the lineage of Δv. Users can program view update strategies using these rules as a design guideline.

Finally, we derive two lemmas for backward/forward update propagation when *put* follows the above put rules and *get* is expressed with monotonic SPJU queries. Let Δs be an update made on source S and Δv be the update on view V derived by *get* from Δs. Also, let $\Delta v'$ be a following update made on view $V \cup \Delta v$ and $\Delta s'$ be the update on source $S \cup \Delta s$ derived by *put* from $\Delta v'$. We derive a lemma for backward update propagation (source update then view update) as follows.

Lemma 1 (backward update propagation). *Let* $get(S \cup \Delta s) = V \cup \Delta v$ *and* $put(S \cup \Delta s, V \cup \Delta v - \Delta v') = S \cup \Delta s - \Delta s'$. *If* $\Delta v' \subseteq \Delta v$ *then* $\Delta s' \subseteq \Delta s$.

Similarly, we derive a lemma for forward update propagation (twice source update).

Lemma 2 (forward update propagation). *Let* $get(S \cup \Delta s) = V \cup \Delta v$ *and* $get(S \cup \Delta s - \Delta s') = V \cup \Delta v - \Delta v'$. *If* $\Delta s' \subseteq \Delta s$ *then* $\Delta v' \subseteq \Delta v$.

These lemmas ensure that if a new update ($\Delta s'$ or $\Delta v'$) is contained in a past update on source (Δs) or view (Δv), the update propagation is localized inside of Δs (backward propagation) or Δv (forward propagation), respectively. These behaviors are realized by tracing the derivation history ($\Delta s \longrightarrow \Delta v$) using lineage and the put rules defined in Definition 1 and 2. By leveraging these features, we safely group affecting records (e.g. Δs and Δv) cascadingly in the Dejima network and put them in the same family record set[7].

4.4 Hybrid mechanism

Since conservative distributed 2PL using family record set prevents distributed deadlock, it is effective when the contention rate is high, but (normal) distributed 2PL is more efficient when the contention rate is low. We propose a hybrid mechanism that appropriately chooses 2PL mode or conservative 2PL mode using family record set depending on the contention rate of transactions. In detail, the hybrid mechanism chooses a preferable mode using speculation; it periodically switches the mode and chooses the better mode by comparing the performance of the two modes. Each participant executes this speculation independently expecting that the mode switch converges in the whole Dejima network.

[6] π^{-1} is an inverse operation of π; default value is assigned for each unprojected attribute.

[7] Note that the lemmas may not hold if we permit user-defined conflict resolution rules (see Section 4.5) or different put rules. As an example, if an insert operation works similarly to that of NoSQL, it may overwrite an exiting record that shares the same key of the insert operation. In this case, family record set does not work as an atomic unit for distributed locking.

4.5 Discussions (for extended version)

Serializablity for join views. In general, an update operation propagates in a tree structure in the Dejima network. When *get* is defined with join operations ($S \bowtie T$), two different updates on source ($\Delta s, \Delta t$) may cause different serialized orders at different participants since these updates do not block each other; some participant receives updates in an order of $\Delta s \longrightarrow \Delta t$ and some other participant may receive updates in the opposite order ($\Delta t \longrightarrow \Delta s$). This behavior still ensures the strong consistency, because these update operations are commutative for join operations. That is $S' \bowtie (T \cup \Delta t) = (S \cup \Delta s) \bowtie T'$ where $S' = S \cup \Delta s$ and $T' = T \cup \Delta t$.

Strong consistency and Eventual consistency. We assume that the conflict resolution is made base on the time preference (a.k.a. first writer's win rule). A possible another design choice is to permit users to write their own (user-defined) conflict resolution rules. Users can design various rules, such as using conflict-free replicated data type (CRDT) [66] or using provenance. Actually, Orchestra takes the latter approach; once a conflict occurs, the system invokes user-defined rules to choose an appropriate version using provenance.

Cyclic update. An update propagation may form cycles in the Dejima network. We permit users to define user-defined merge function how to handle different versions caused by the cycles. There can be two possible cases of forming cycles: 1) the cyclic update is designed by users so they choose the newer version or merge versions and continue update propagation, or 2) the cyclic update is exceptional and needs to be handled as error. Also, there are two types of cycle shape: 1) a client update spawns multiple paths of the update propagation and those paths cross at the same participant, or 2) a client update spawns a single path and the update propagates back to the original participant. We detect cycles using the lineage of update.

5 SYSTEM IMPLEMENTATION

Due to the page limitation, we present a summary of the system implementation of the Dejima architecture. The details are provided in Appendix A of the extended version of this paper, which can be found at [1] (also in the supplementary file). The source code of the Dejima prototype has also been released at [1].

In our implementation, each participant consists of a DBMS and an API server. PostgreSQL is used for the DBMS. The API server synchronizes the update on the Dejima tables in the same Dejima group, and is implemented using the Falcon Web Framework [2]. Update propagation is implemented using triggers, as described in Section 3. To start a transaction, once a participant's base table is updated, the participant requests a *local lock* from its own DBMS. If the local lock acquisition is successful, the participant communicates other participants via the API server, and requests a *global lock* from their DBMSs for the affected records with the same lineage, using family record set terminology introduced in Section 4. If successful, a trigger is activated and it transforms the `bt` of Provider *A*

V	L	D	R
1	120	1765	1
2	3866	5228	2
3	6545	6545	0

`bt` of Provider *B*

V	L	D	R	AL1	AL2
1	6201	6201	0	TRUE	TRUE
2	4138	1947	3	TRUE	FALSE
3	1693	1693	0	FALSE	TRUE

`bt` of Provider *C*

V	L	D	R	type	AL2
1	5288	5288	0	sedan	TRUE
2	367	4682	5	SUV	TRUE
3	2659	2659	0	wagon	FALSE

Figure 3: Vehicle databases.

(a) `mt` of Alliance 1.

V	L	D	R	P
1	120	1765	1	A
2	3866	5228	2	A
3	6545	6545	0	A
1	6201	6201	0	B
2	4138	1947	3	B

(b) `mt` of Alliance 2.

V	L	D	R	P
1	6201	6201	0	B
3	1693	1693	0	B
1	5288	5288	0	C
2	367	4682	5	C

Figure 4: Tables of alliances.

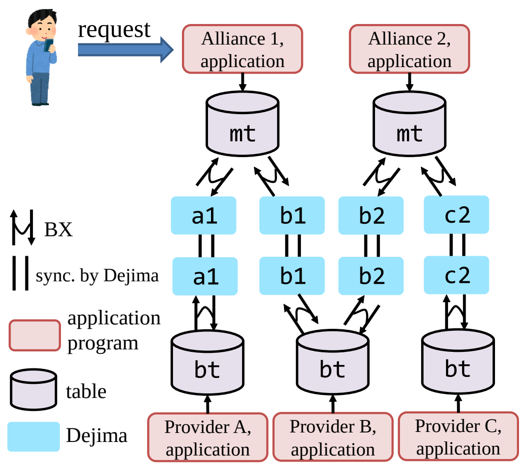


Figure 5: Dejima architecture for ride-sharing alliances.

(a) Dejima table `a1`.

V	L	D	R
1	120	1765	1
2	3866	5228	2
3	6545	6545	0

(b) Dejima table `b1`.

V	L	D	R
1	6201	6201	0
2	4138	1947	3

Figure 6: Dejima tables corresponding to Figures 3 and 4.

update made on the base table to the update on the Dejima table. The update is sent to the API server of this participant and replicated to the API servers of other participants in the same Dejima group. The received update is transformed to the update on base tables and applied to these participants' DBMSs. Dejima tables are updatable views and they are derived from the view update strategy and defined using BIRDS [71, 72], a programming framework based on Datalog. View definitions are then translated to equivalent SQL programs, along with PL/pgSQL triggers and procedures to perform bidirectional view updates.

6 CASE STUDY

To show the usefulness of Dejima architecture in practice, we present an application scenario that corresponds to the example of ride-sharing alliances shown in Section 1.

As a first case, we will use vehicle data integration for a ride-sharing alliance. We have actually created a demonstration program using Dejima, according to the example of two alliances and three providers illustrated in Figure 1. The following explanation is based on it.

Let P_A, P_B, and P_C denote the providers *A*, *B*, and *C*, respectively. Let us assume that each provider has base table `bt` for managing vehicles. Figure 3 illustrates an example of these tables. Note that the local schemas of the providers are different. In each table, attributes `V`, `L`, and `D` denote the ID, the current location, and the current destination of each vehicle, respectively. Attribute `R` denotes the ID of the request assigned to each vehicle. An empty vehicle has `R` $= 0$. Although a ride-sharing service requires various kinds of data other than vehicles, we simplify them as much as possible to focus on vehicle data. Therefore, we omit the information about requests and passengers here, and assume any positive value of `R` (i.e. any request) never appear twice in the tables. In addition, each value for `L` (and `D`) is simplified as an integer here. For example, we can consider them to be nodes in the road network that are commonly used by all providers.

Ideally, ride-sharing alliances should allow each provider to determine which vehicle in it is available on the alliances in a flexible and easy way. In addition, when a vehicle is assigned to a request by an alliance, the update of data should be reflected in the corresponding provider. The Dejima architecture enables this by solving the inflexible data integration problem and the read-only views problem explained in Section 1. For example, let us assume that P_A wants to publish its all vehicles in the alliance 1, and P_B wants to publish every vehicle whose attribute `AL1` (or `AL2`) is 'True' to the alliance 1 (or 2, respectively). Similarly, we assume that P_C wants to publish every vehicle whose attribute `AL2` is 'True' to the alliance 2. Then, the base tables (named `mt`) of the Alliances 1 and 2 should be those illustrated in Figure 4.

Table (a) in Figure 4 is the data of vehicles published by P_A and P_B to the Alliance 1. Note that the new attribute `P` is added for indicating the ID of the provider to which each vehicle belongs. Similarly, Table (b) is the data of P_B and P_C for the Alliance 2.

To realize the above data integration, we establish a Dejima architecture as Figure 5, where base tables are shown in gray and Dejima tables are shown in cyan. Due to space limitations, we explain our implementation of the Alliance 1 only. In the Dejima architecture, this is achieved by writing BIRDS codes [71, 72] (based on Datalog) illustrated in Figures 7 to 9. P_A (or P_B) and the Alliance 1 share the data through Dejima table `a1` (or Dejima table `b1`, respectively). Figure 6 illustrates the data of Dejima table `a1` and Dejima table `b1` for the data illustrated in Figures 3 and 4.

P_A should write the code in Figure 7 to publish its vehicles to Dejima table a1 and what should be done if a1 is updated by the Alliance 1. The first two lines tell the schema of bt and a1 to BIRDS. The line a1(V,L,D,R) :- bt(V,L,D,R) indicates that P_A publish its all vehicles to a1. The last two lines indicate the following two update strategies when a1 is updated by the Alliance 1: (1) if the update deletes a record that also exists in bt, then the record has to be deleted on bt, and (2) if the update inserts a record that does not exist in bt, then the record has to be inserted into bt.

Similarly, P_B should write the code in Figure 8 for Dejima table b1. While this code is more complicated than the previous code, we can see that P_B can publish its vehicles according to the assumption explained above. Furthermore, the update strategies (the last two lines) enable P_B to reflect the update of b1 by the Alliance 1 in its base table bt correctly. Concretely, when the Alliance 1 updates a vehicle on b1 by setting new values of L, D, or R, the update is reflected in the vehicle in bt of P_B.

On the other hand, the Alliance 1 should write the code in Figure 9 for Dejima table a1. The last two lines indicate update strategies when a1 is updated by P_A. For example, when a record with vehicle ID of V in a1 is deleted, a record having the same ID in mt is also deleted if its value of P is 'A'. Note that we omit the code for Dejima table b1 of the Alliance 1 because it is almost identical to this code.

With these codes converted to PostgreSQL by BIRDS, the Dejima architecture can automatically achieve the data integration for these alliances and providers. As we have seen, each provider does not have to abandon its local schema and change it to a global schema. We have also seen that data publishing policy and update strategy can be built flexibly. For example, P_C can "publish vehicles whose type is sedan or SUV", instead of using AL2 as above, by writing the BIRDS code appropriately.

With the Dejima architecture, when an alliance makes a request assignment in its own base table, it is automatically reflected in the provider's base table, and when a provider updates its vehicle data, it is automatically reflected in the alliance. Therefore, once this data integration is complete, each alliance and provider can create applications without worrying about the base tables of others. In the following, we will describe the outline of our demo application according to Figure 5.

Let us consider a passenger sending a request to Alliance 1. At this time, the passenger's application informs the Alliance 1 application (hereafter referred to as App1) of the ID of the request and his/her current location. App1 performs the assignment procedure by the following transaction.

(1) Using PostgreSQL, it selects all available vehicles (R = 0) in its base table mt.
(2) From among them, find the vehicle closest to the passenger's current location (using the road network data).
(3) Using PostgreSQL, update R of that vehicle to the passenger's request ID and D to the passenger's current location on mt.

If this transaction is successful, it means that the provider's base table with that vehicle has also been updated correctly by the Dejima architecture.

Each provider can configure PL/pgSQL triggers that work when its base table bt is updated via Dejima to create a data-driven application. For example, a provider could send a command to that vehicle to go to the passenger's current location. In our demonstration, we only simulate the movement of vehicles because we cannot use real vehicles. The vehicle would travel to the passenger's current location using the shortest path on the road network, and then update D to the passenger's destination and travel to it. Note that the location L of the vehicle is updated on bt whenever the vehicle passes any vertex on the road network. Of course, these updates propagate to the alliance by the Dejima architecture. The passenger's destination is set randomly in our demo, although he/she would tell the driver his destination when he/she gets on a real vehicle.

```
source bt('V':int,'L':int,'D':int,'R':int).
view a1('V':int,'L':int,'D':int,'R':int).
%view definition
a1(V,L,D,R) :- bt(V,L,D,R).
%update strategies
-bt(V,L,D,R) :- bt(V,L,D,R), NOT a1(V,L,D,R).
+bt(V,L,D,R) :- NOT bt(V,L,D,R), a1(V,L,D,R).
```

Figure 7: BIRDS code between Provider A and Dejima table a1.

```
source bt('V':int,'L':int,'D':int,'R':int,
         'AL1':string,'AL2':string).
view b1('V':int,'L':int,'D':int,'R':int).
%view definition
b1(V,L,D,R) :- bt(V,L,D,R,AL1,AL2), AL1='True'.
%update strategies
-bt(V,L,D,R,AL1,AL2) :- bt(V,L,D,R,AL1,AL2),
                        NOT b1(V,L,D,R), AL1='True'.
+bt(V,L,D,R,AL1,AL2) :- NOT bt(V,L,D,R,AL1,_), b1(V,L,D,R),
                        bt(V,_,_,_,AL1,AL2), AL1='True'.
```

Figure 8: BIRDS code between Provider B and Dejima table b1.

Readers might consider the transaction introduced above is problematic. Recall that P_B publishes a vehicle to both the alliances. Therefore, if both the alliances start transactions to assign the vehicle simultaneously, then either transaction has to fail and rollback to prevent data inconsistency such as a double-booking. Because the Dejima architecture guarantees the global consistency and controls such processes automatically, application programmers do not need to worry about anything else. In order to guarantee global consistency, conventional architectures require the centralized design that manages all the data of both alliances in one place. However, with the Dejima architecture, alliance databases and applications can be deployed on various participants. For example, a mediator can be created for each alliance, or a provider can have a database for each alliance it participates in.

7 EXPERIMENTS

This section reports our experimental study to evaluate Dejima's performance on transaction management. In the experiments, we use the m4.2xlarge instance type, which has 8 CPU cores and 32GB RAM, on Amazon EC2. We deploy a proxy server and a DBMS server

```
source mt('V':int,'L':int,'D':int,'R':int,'P':string).
view a1('V':int,'L':int,'D':int,'R':int).
%update strategies
-mt(V,L,D,R,P) :- mt(V,L,D,R,P), NOT a1(V,L,D,R), P='A'.
+mt(V,L,D,R,P) :- NOT mt(V,L,D,R,P), a1(V,L,D,R), P='A'.
```

Figure 9: BIRDS code between Alliance 1 and Dejima table `a1`**.**

Table 1: Benchmark parameter setting.

Parameter	Values
# of participants	20, 40, 60, 80, 100
Propagation ratio	20%, 40%, 60%, 80%, 100%
Skew factor (only YCSB)	0.2, 0.4, 0.6, 0.8, 0.99

within one instance using Docker. So one server corresponds to one participant.

7.1 Experiment Setup

7.1.1 Benchmarks. We use two benchmarks, YCSB and TPC-C, which have been widely used for DBMS performance evaluation. They are modified for the evaluation in a decentralized environment with relatively high and low contentions, respectively. The details of the benchmarks are as follows.

YCSB. We use workload A (update heavy) in the YCSB benchmark [3] and create transactions that consist of ten queries (five reads and five writes) in workload A. All participants have a single base table with a primary key and ten columns following the original YCSB benchmark. The record ID in each query is generated by a Zipfian distribution with a varying skew factor. In addition, we add ten columns to the base table with random integer values between 0 and 99, named `cond1`, `cond2`, . . ., `cond10` for controlling the propagation ratio of queries. Before measuring the performance, we insert records to each participant's base table and each record is propagated according to the global schema (see the network configuration paragraph below). There are totally 100,000 records in the network and records are distributed evenly to participants.

TPC-C. We use `New-Order` and `Payment` transactions in the original TPC-C benchmark, executed in a 1:1 ratio. No other transactions are executed. All participants have nine tables, following the original TPC-C benchmark. We use the `customer` table as base table. We assume that each participant corresponds to a `district` in the benchmark. Ten columns are added to the `customer` table for controlling the propagation ratio (same as YCSB). Before measuring the performance, we insert records to each participant. At first, we insert records to `warehouse`, `district`, `stock`, and `items` in each participant, following the original TPC-C benchmark. Then, we insert 3,000 records to `customer`, and records corresponding to these customers are added to other tables. Each record is propagated to other participants according to the global schema.

Network Configuration. For network topology, we use a randomly generated spanning tree. Each edge in the spanning tree corresponds to a Dejima group of two participant, and is assigned with a label in {`cond1`, `cond2`, . . ., `cond10`} at random. Figure 10 shows an example of a network with 20 participants. The definition

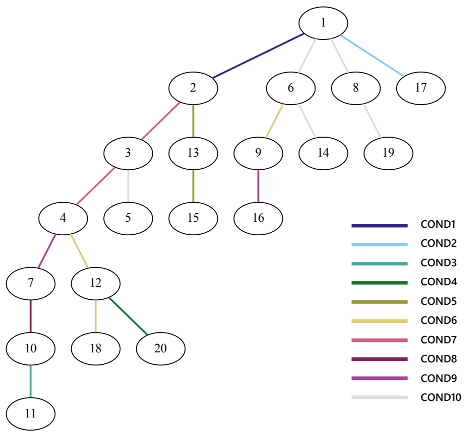


Figure 10: An example of network topology.

of each Dejima table is a selection of all the columns of a base table using a `WHERE` clause, which is determined by the label assigned to the edge (e.g., "`SELECT * FROM base_table WHERE cond1 <` x", where x indicates the propagation ratio in the range of [0, 1]).

7.1.2 Metrics. We evaluate throughput as the performance of transaction methods. We report the number of committed transactions for 1,200 seconds per participant. We vary the number of participants and the propagation ratio in the two benchmarks. We also vary the skew factor in YCSB. Table 1 shows the parameter setting for the two benchmarks. In YCSB, we set 60, 100%, and 0.8 as the default number of participants, propagation ratio, and skew factor, respectively. In TPC-C, we set 60 and 80% as the default number of participants and propagation ratio, respectively.

7.1.3 Methods. We evaluate the performance of three methods: **FRS-based C2PL**, **2PL**, and **Hybrid**. FRS-based C2PL is our method proposed in Section 4. 2PL is a strict two-phase locking method. In 2PL, the lock of a participant for updated records is acquired only when the update is propagated to the participant, and then the update is executed if acquisition is successful. We use the NO WAIT strategy to prevent deadlocks. Transactions are committed following to the two-phase commit protocol as well as our proposed method. In the Hybrid method, each participant periodically measures the performance of FRS-based C2PL and 2PL for 30 seconds once every 300 seconds, and then switches to the method with better throughput. We run the Hybrid method for 600 seconds before measuring the performance. Then, we disable the switch for 1,200 seconds and measure the throughput.

7.2 Experimental Results

7.2.1 YCSB. We first show the performance on YCSB.

Table 2 shows the throughput when we vary the number of participants on YCSB. From this result, we can see that our proposed methods (FRS-based C2PL and Hybrid) consistently outperform 2PL, indicating that our methods work well on a workload with relatively high contention. Moreover, the performance gaps between

Table 2: Throughput on YCSB, varying # of participants.

Method	# of participants				
	20	40	60	80	100
FRS-based C2PL	24.55	17.31	12.03	8.94	6.78
Hybrid	24.62	17.46	11.60	8.67	6.49
2PL	24.07	16.06	11.36	8.22	4.87

Table 3: Throughput on YCSB, varying propagation ratio.

Method	Propagation ratio [%]				
	20	40	60	80	100
FRS-based C2PL	788.52	190.39	50.02	22.69	12.40
Hybrid	958.47	210.37	53.33	23.45	12.28
2PL	1110.44	215.97	53.23	23.47	11.62

our methods and 2PL become larger (up to 1.4x) as the number of participants increases. This is because as the number of participants increases, the update caused by a transaction is propagated to more participants, and this increases the execution time of the transaction. This results in more frequent aborts for 2PL when executing the transaction, while for FRS-based C2PL and Hybrid, the effect of abort is not so significant because transactions are not executed until global locks are acquired. The Hybrid method performs slightly worse than FRS-base C2PL. This is because not all participants can choose the most suitable protocol in the 600 seconds of pre-execution.

Table 3 shows the throughput when we vary the propagation ratio on YCSB. From this result, we can see that the performance of 2PL is better than FRS-based C2PL and Hybrid when the propagation ratio is low, and the gap increases when we move towards a lower propagation ratio. This is because when the propagation ratio is low, the number of participants affected by a transaction is small, and this reduces the number records shared among multiple participants as well as contention. This result suggests users should resort to a traditional 2PL protocol for low propagation ratios. For Hybrid, since each participant can choose a better protocol for the workload, its outperforms FRS-based C2PL at low propagation ratio.

Table 4 shows the throughput on YCSB when we vary the skew factor of the Zipfian distribution used to generate record IDs. From this result, we can see that FRS-based C2PL and Hybrid achieve better performance than 2PL when the skew factor is greater than 0.6. This is because a larger skew factor indicates that the generated record IDs are more concentrated to a small range, and thus it results in higher contention of transactions. In this case, FRS-based C2PL and Hybrid are able to improve the performance by reducing the abort of ongoing transactions when distributed deadlocks occur frequently.

7.2.2 TPC-C Results. Next, we show the performance on TPC-C.

Table 5 shows the throughput when we vary the number of participants on TPC-C. From this result, we can see that FRS-based C2PL is inferior to 2PL, and the gap between the two is roughly 1.1x. This is because distributed deadlocks do not occur frequently in this workload with low contention. This results in small number of aborts while the overhead of executing FRS-based C2PL negatively

Table 4: Throughput on YCSB, varying skew factor.

Method	Skew factor				
	0.2	0.4	0.6	0.8	0.99
FRS-based C2PL	11.67	11.75	11.67	12.40	12.12
Hybrid	12.83	12.71	12.38	12.28	11.69
2PL	14.27	14.19	13.73	11.62	8.11

Table 5: Throughput on TPC-C, varying # of participants.

Method	# of participants				
	20	40	60	80	100
FRS-based C2PL	55.74	36.71	27.20	17.30	11.17
Hybrid	63.49	39.06	29.81	20.41	12.38
2PL	63.04	39.91	29.27	20.61	12.31

Table 6: Throughput on TPC-C, varying propagation ratio.

Method	Propagation ratio [%]				
	20	40	60	80	100
FRS-based C2PL	105.73	35.80	20.91	19.38	19.47
Hybrid	172.84	39.32	23.44	18.99	20.05
2PL	182.66	41.53	25.13	20.42	19.43

affects the performance. Hybrid delivers similar performance to 2PL, and outperforms 2PL in a few cases, because each participant is able to autonomously choose the better protocol to deal with different level of contention.

Table 6 shows the throughput when we vary the propagation ratio on TPC-C. From this result, we can see that FRS-based C2PL has similar performance to 2PL at high propagation ratio, but the performance deteriorates as we move towards low propagation ratio, similar to what we have witnessed on YCSB. In addition, the deteriorationis even worse on TPC-C than YCSB, due to the low contention of TPC-C. Hybrid's performance is close to 2PL, because it is able to select a appropriate protocol at different level of contention, hence addressing the drawback of FRS-based C2PL at low propagation ratio.

7.2.3 Summary. The above results suggest that FRS-based C2PL efficiently performs distributed transaction management on a workload with high contention, while 2PL's performance in this case is inferior due to distributed deadlocks. On the other hand, when the workload contains low contention, the overhead for adopting FRS-based C2PL negatively affects the performance. Nonetheless, the Hybrid method can overcome the performance deterioration of FRS-based C2PL by letting each participant choose an appropriate protocol. As such, we recommend users to choose FRS-based C2PL for high contention workloads and Hybrid or 2PL for low contention workloads.

8 RELATED WORK

8.1 View Update

The view update problem is a classical problem that has a long history in database research [12, 23, 24, 50, 54, 55, 58–60]. It was realized very early that the translation from view updates to source updates may not always exist, and even if it does exist, it may not be unique [23, 24]. To solve the ambiguity of translating view updates to updates on base relations, some works proposed the concept of view complement [12, 54], some others allowed users to choose the one through an interaction [50, 55] or select relevant update policies based on user intentions [58–60]. Bohannon et al. [14] employed BX to solve the view update problem in relational databases and proposed a bidirectional language, called relational lenses, by enriching the SQL expression for defining views of projection, selection, and join. A recent work [39] proposes an incrementalization technique for optimizing relational lenses. Melnik et al. [62] proposed a novel declarative mapping language for specifying the relationship between application entity views and relational databases. User-specified mappings are validated and compiled into round-tripping bidirectional views for the view update translation. Salem et al. [65] proposed an algorithm for how views/replicas can be lazily maintained using delta relations. Other works on update propagation (e.g., by lazy maintenance) for database replication include [5, 15, 22]. Besides, the case of view maintenance in a warehousing environment was studied in [40].

8.2 Distributed Transaction Management

There have been related techniques in replicate database systems that perform distributed transactional processing without concurrency control by executing transactions serially, such as [45, 68, 69]. Most of these techniques require to use the two-phase commit protocol (2PC). Instead of using 2PC, Calvin [69] makes the replicated databases consistent by using one-round trip protocol during distributed transaction execution. The idea of Calvin is that we can remove the effects of nondeterministic events (such as node failures) by employing consistent replication techniques, so it can skip the 2nd phase of 2PC.

In the cloud environments, Google's Megastore [11] and Spanner [19] are well-known systems that achieve strong consistency between data replications. Megastore employs Paxos to make strong consistency between data replications. Spanner utilizes TrueTime, a distributed clock with bounded uncertainty, so that it archives efficient distributed transactions by assigning timestamps to transactions without global coordination. Those systems are based on the traditional non-deterministic transaction ordering.

There is a new approach to ensure global consistency without using 2PC. Coordination avoidance [10, 74] permits the users to write application-level invariant confluence and then the system generates a necessary and sufficient conditions for invariant-preserving and coordination-free execution of transactions. The coordination avoidance is promising approach, however, it is not easy for the users to write consistent invariant confluence.

8.3 Data Sharing and Data Interoperability

The classical architecture of data integration is centralized. That is, one mediator gathers all the distributed data, transforms the data according to the schema mappings, and provides the uniform data to its users. On the other hand, decentralized data integration, or peer-to-peer data integration, has been focused on and many prototype systems have been developed in the last two decades.

Piazza [36, 37] is one of the first projects on decentralized data integration. Designed for integrating distributed XML documents, it provides query answering functionality based on the certain answer semantics by rewriting given query. However, updating XML documents on peers is out of its scope.

Orchestra [42, 49], a successor project of Piazza, proposed a novel concept CDSS (collaborative data sharing systems) motivated by the need for collaborative sharing of scientific data, which are produced by independent researchers without any global agreement. In CDSS, data inconsistency between different peers is positively allowed so that different scientific groups can treat different scientific data on the same object, experiment, etc. Therefore, transaction processing over different peers is not realized in CDSS.

The Hyperion project [7, 51] proposes an architecture of a *peer database management system* (PDBMS for short). A PDBMS consists of three components: an interface to the users, an ordinary DBMS, and a P2P layer, which is the key component of a PDBMS. A P2P layer has the following three functionalities: managing neighbor peer relationship, query rewriting for answering queries, and enforcing data consistency upon different peers. Because successive query rewriting loses information of the original query, a framework called GrouPeer [47, 48] was proposed to improve the quality of answering queries. It finds semantically similar peers and makes them neighbors to avoid successive query rewriting. In these frameworks, it is unclear whether updating data on other peers is possible. Even if this is the case, controlling update strategy does not seem to be supported. Other related studies on P2P systems focused on data replication [18, 34, 57], while [6] studied P2P systems with transactional semantics.

Garlic [46] is a prototype based on DB2 and extends its ability to federate relational data sources. It features DB2 catalogs that can be used as an extensible repository for the metadata needed to access remotely-stored information, and a set of solutions were proposed to address query planning and execution in the federated environment. Other related studied in this category include schema management and provenance. We refer readers to [32, 75], in which a framework that uses data examples as the basis for understanding and refining schema mappings and a suite of provenance-based tools that supports the debugging and tracing of schema mappings and transformation queries were proposed, respectively.

9 CONCLUSION

In this paper, we proposed the Dejima architecture, which processes transaction management for data integration. The architecture is designed based on a decentralized paradigm. By employing updatable views which are realized using bidirectional transformation, the Dejima architecture integrates databases in a flexible manner, so that each participant can autonomously manage its own database and collaborate with other participants. Participants are organized into Dejima groups in a network, and exchange data through Dejima tables specified by a global schema. Global consistency is guaranteed through distributed transaction management. A case study was discussed using an application scenarios for ride-sharing alliance. We implemented a prototype of the Dejima architecture and evaluated its transaction management performance using the YCSB and TPC-C benchmarks modified for decentralized data integration.

REFERENCES

[1] The Dejima Prototype. https://github.com/ekayim/dejima-prototype.

[2] The Falcon Web Framework. https://falcon.readthedocs.io/en/stable/.

[3] The YCSB Benchmark. https://github.com/brianfrankcooper/YCSB.

[4] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan, et al. The seattle report on database research. *ACM SIGMOD Record*, 48(4):44–53, 2020.

[5] D. Agrawal, A. E. Abbadi, and K. Salem. A taxonomy of partitioned replicated cloud-based database systems. *IEEE Data Eng. Bull.*, 38(1):4–9, 2015.

[6] S. Antony, D. Agrawal, and A. E. Abbadi. P2P systems with transactional semantics. In A. Kemper, P. Valduriez, N. Mouaddib, J. Teubner, M. Bouzeghoub, V. Markl, L. Amsaleg, and I. Manolescu, editors, *EDBT*, volume 261 of *ACM International Conference Proceeding Series*, pages 4–15. ACM, 2008.

[7] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion project: from data integration to data coordination. *ACM SIGMOD Record*, 32(3):53–58, 2003.

[8] Y. Asano, D. Herr, Y. Ishihara, H. Kato, K. Nakano, M. Onizuka, and Y. Sasaki. Flexible framework for data integration and update propagation: System aspect. In *SFDI@BigComp*, pages 1–5, 2019.

[9] Y. Asano, S. Hidaka, Z. Hu, Y. Ishihara, H. Kato, H. Ko, K. Nakano, M. Onizuka, Y. Sasaki, T. Shimizu, V. Tran, K. Tsushima, and M. Yoshikawa. Making view update strategies programmable - toward controlling and sharing distributed data -. *CoRR*, abs/1809.10357, 2018.

[10] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3):185–196, 2014.

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

[12] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[13] V. Bárány, B. ten Cate, and M. Otto. Queries with guarded negation. *PVLDB*, 5(11):1328–1339, 2012.

[14] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: A language for updatable views. In *PODS*, pages 338–347, 2006.

[15] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD*, pages 97–108. ACM Press, 1999.

[16] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.

[17] A. Cleve, E. Kindler, P. Stevens, and V. Zaytsev. Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491). *Dagstuhl Reports*, 8(12):1–48, 2019.

[18] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In M. Mathis, P. Steenkiste, H. Balakrishnan, and V. Paxson, editors, *SIGCOMM*, pages 177–190. ACM, 2002.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[20] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, pages 260–283, 2009.

[21] C. N. G. Dampney and M. Johnson. Half-duplex interoperations for cooperating information systems. In *Advances in Concurrent Engineering*, pages 565–571, 2001.

[22] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In Z. M. Özsoyoglu and S. B. Zdonik, editors, *ICDE*, pages 424–435. IEEE Computer Society, 2004.

[23] U. Dayal and P. A. Bernstein. On the updatability of relational views. In *VLDB*, pages 368–377, 1978.

[24] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

[25] Z. Diskin. Algebraic models for bidirectional model synchronization. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *MODELS*, pages 21–36, 2008.

[26] Z. Diskin. Update Propagation Over a Network: Multi-ary Delta Lenses, Tiles, and Categories, 2019. Keynote talk of the 3rd Workshop on Software Foundations for Data Interoperability (SFDI2019+).

[27] Z. Diskin and S. Hidaka. Personal communications, Oct. 2019.

[28] Z. Diskin, H. König, and M. Lawford. Multiple model synchronization with multiary delta lenses. In *FASE*, pages 21–37, 2018.

[29] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration.* Morgan Kaufmann, 2012.

[30] S. Fischer, Z. Hu, and H. Pacheco. A clear picture of lens laws. In *MPC*, pages 215–223, 2015.

[31] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

[32] B. Glavic, G. Alonso, R. J. Miller, and L. M. Haas. TRAMP: understanding the behavior of schema mappings through provenance. *PVLDB*, 3(1):1314–1325, 2010.

[33] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[34] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In H. Jacobsen, editor, *Middleware*, volume 3231 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2004.

[35] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[36] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):787–798, 2004.

[37] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for Semantic Web applications. In *WWW*, pages 556–567, 2003.

[38] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *POPL*, pages 371–384, 2011.

[39] R. Horn, R. Perera, and J. Cheney. Incremental relational lenses. *PACMPL*, 2(ICFP):74:1–74:30, 2018.

[40] N. Huyn. Multiple-view self-maintenance in data warehousing environments. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB*, pages 26–35. Morgan Kaufmann, 1997.

[41] Y. Ishihara, H. Kato, K. Nakano, M. Onizuka, and Y. Sasaki. Toward BX-based architecture for controlling and sharing distributed data. In *SFDI@BigComp*, pages 1–5, 2019.

[42] Z. G. Ives, N. Khandelwal, A. Kapur, and M. Cakir. ORCHESTRA: rapid, collaborative sharing of dynamic data. In *CIDR*, pages 107–118, 2005.

[43] M. Johnson and R. D. Rosebrugh. Spans of lenses. In K. S. Candan, S. Amer-Yahia, N. Schweikardt, V. Christophides, and V. Leroy, editors, *CEUR@EDBT/ICDT*, pages 112–118, 2014.

[44] M. Johnson and R. D. Rosebrugh. Cospans and symmetric lenses. In S. Marr and J. B. Sartor, editors, *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 21–29, 2018.

[45] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, pages 603–614, 2010.

[46] V. Josifovski, P. M. Schwarz, L. M. Haas, and E. T. Lin. Garlic: a new flavor of federated query processing for DB2. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD*, pages 524–532. ACM, 2002.

[47] V. Kantere, D. Bousounis, and T. Sellis. GrouPeer: A system for clustering PDMSs. *PVLDB*, 4(12):1371–1374, 2011.

[48] V. Kantere, D. Tsoumakos, T. Sellis, and N. Roussopoulos. GrouPeer: Dynamic clustering of P2P databases. *Information Systems*, 34(1):62–86, 2009.

[49] G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen. Collaborative data sharing via update exchange and provenance. *ACM Trans. Database Syst.*, 38(3):19:1–19:42, 2013.

[50] A. M. Keller. Choosing a view update translator by dialog at view definition time. In *VLDB*, pages 467–474, 1986.

[51] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *SIGMOD*, pages 325–336, 2003.

[52] H. Ko and Z. Hu. An axiomatic basis for bidirectional programming. *PACMPL*, 2(POPL):41:1–41:29, 2018.

[53] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, pages 87–98, 2010.

[54] R. Langerak. View updates in relational databases with an independent scheme. *ACM Trans. Database Syst.*, 15(1):40–66, 1990.

[55] J. A. Larson and A. P. Sheth. Updating relational views using knowledge at view definition and view update time. *Information Systems*, 16(2):145–168, 1991.

[56] P. J. Leach, R. Salz, and M. H. Mealling. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, 2005.

[57] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In K. Ebcioglu, K. Pingali, and A. Nicolau, editors, *ICS*, pages 84–95. ACM, 2002.

[58] Y. Masunaga. A relational database view update translation mechanism. In *VLDB*, pages 309–320, 1984.

[59] Y. Masunaga. An intention-based approach to the updatability of views in relational databases. In *IMCOM*, pages 13:1–13:8, 2017.

[60] Y. Masunaga, Y. Nagata, and T. Ishii. Extending the view updatability of relational databases from set semantics to bag semantics and its implementation on PostgreSQL. In *IMCOM*, pages 19:1–19:8, 2018.

[61] L. Meertens. Designing constraint maintainers for user interaction. Manuscript available at http://www.kestrel.edu/home/people/meertens, 1998.

[62] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4):22:1–22:50, 2008.

[63] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: a P2P-based system for distributed data sharing. In *ICDE*, pages 633–644, 2003.

[64] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., 2nd edition, 1999.

[65] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How to roll a join: Asynchronous incremental view maintenance. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *SIGMOD*, pages 129–140. ACM, 2000.

[66] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400. Springer LNCS volume 6976, Oct. 2011.

[67] P. Stevens. Bidirectional transformations in the large. In *MODELS*, page 1–11, 2017.

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

[69] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[70] M. Tom. A visual guide to the twisted web created by the Uber/Didi merger. https://pitchbook.com/news/articles/a-visual-guide-to-the-twisted-web-created-by-the-uberdidi-merger, 2016.

[71] V.-D. Tran, H. Kato, and Z. Hu. BIRDS | Bidirectional Transformation for Relational View Update Datalog-based Strategies. available at: https://dangtv.github.io/BIRDS/, 2019.

[72] V.-D. Tran, H. Kato, and Z. Hu. Programmable view update strategies on relations. *PVLDB*, 13(5):726–739, 2020.

[73] J. D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40, 1997.

[74] M. Whittaker and J. M. Hellerstein. Interactive checks for coordination avoidance. *PVLDB*, 12(1):14–27, 2018.

[75] L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In S. Mehrotra and T. K. Sellis, editors, *SIGMOD*, pages 485–496. ACM, 2001.

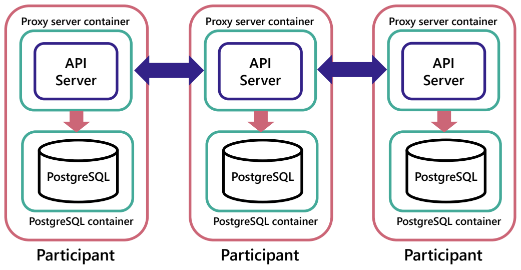


Figure 11: Overview of Dejima implementation.

A IMPLEMENTATION DETAILS

Figure 11 shows the implementation of the Dejima architecture. Each participant has a DBMS and an API server. PostgreSQL 11.2 is used to implement the DBMS. The API server synchronizes the update on the Dejima tables in the same Dejima group, and is implemented using the Falcon Web Framework [2]. Database servers and API servers are created using Docker containers.

Update propagation in the Dejima architecture is realized by sharing BX-based updatable views. We utilize BIRDS [71, 72], a programming framework based on Datalog, to implement update propagation. View definitions are derived from the view update strategy and translated to equivalent SQL programs. PL/pgSQL triggers and procedures are output to perform bidirectional view updates. In addition, we use delta relations [33] for incremental view maintenance.

Next we show the implementation details of transaction management in Dejima architecture.

A.1 Starting a Transaction

When a user submits a transaction to a participant, the API server that accepts the transaction first generates a global transaction ID by combining its own participant ID and the identifier generated using the UUID [56]. The API server also obtains a database connection from the database connection pool and stores the pair of (global transaction ID, database connection).

A.2 Acquiring Local Locks

The API server extracts the `SELECT`, `UPDATE`, and `DELETE` statements from the input transaction, and generates a query for local lock (i.e., to lock the affected records in the participant's own DBMS) acquisition using the `WHERE` clause of each statement. As shown in Figure 12, if there is a `SELECT` statement in the transaction, the API server issues a `SELECT` statement with the extracted `WHERE` clause and the `FOR SHARE` keyword to the database server to obtain a local lock. If there is an `UPDATE` or `DELETE` statement, then `FOR UPDATE` is used instead of `FOR SHARE`, and the `SELECT` statement for lock acquisition requests to obtain the lineage of the corresponding record. If the local lock acquisition fails, the transaction is aborted.

A.3 Acquiring Global Locks

The acquisition of global locks (i.e., to lock the affected records with the same lineage in the network) is shown in Figure 13. If the acquisition of a local lock is successful, the participant's API

```
SELECT col1 FROM basetable WHERE col2 = 1
UPDATE basetable SET col3 = 2 WHERE col4 = 3
```

Transaction

```
SELECT * FROM basetable WHERE col2 = 1 FOR SHARE
SELECT lineage FROM basetable WHERE col4 = 3 FOR UPDATE
```

Queries for locking record and getting lineages

Figure 12: Examples of SQL statement for local lock acquisition.

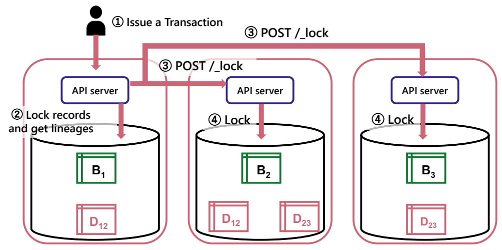


Figure 13: Implementation of global lock acquisition.

server sends the global transaction ID and the acquired lineage information to other participants, and requests a lock on a record with the same lineage. The information is transmitted in JSON format and sent to the /_lock endpoint of each participant using the `POST` method. Each participant that accepts the request obtains a database connection from the database connection pool and stores the pair of (global transaction ID, database connection) sent from participant that requests the lock. Next, a `SELECT` statement with the `FOR UPDATE` keyword is used to acquire a lock on the record with the corresponding lineage. Depending on the result of the lock acquisition, acknowledgement (Ack) or negative-acknowledgment (Nak) is returned to the participant that requests the lock. If the requesting participant does not receive an Ack from all participants, it releases the acquired lock and aborts the transaction. The acquired lock can be released by sending the data describing the global transaction ID in JSON format to the /_unlock endpoint of each participant using the `POST` method.

A.4 Update Propagation

Figure 14 shows the implementation of update propagation. If all the participants reply Ack for lock acquisition, we execute the input transaction. Then, since w employ delta relations [33] for incremental view maintenance, we execute two procedures defined by BIRDS: (1) propagate the update from base table to Dejima table, and (2) get the delta in Dejima table in JSON format. Figure 15 shows an example of delta in JSON format. As shown in the figure, the delta consists of the name of the Dejima table to be updated, the records to be inserted into the Dejima table, and the records to be deleted from the Dejima table.

Next, the delta and the global transaction ID in JSON format are sent to the /_propagate endpoint of each participant in the same Dejima group using the `POST` method. Upon receiving this information, each participant retrieves the corresponding database

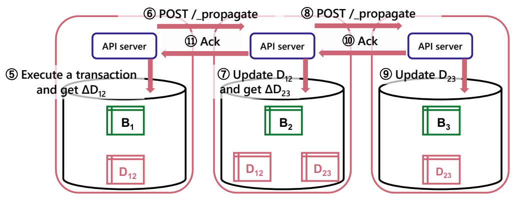


Figure 14: Implementation of update propagation.

```
{
        'dejima_table' : dejima_table,
        'insertion' :
                [ {'ID': 1, 'col1':2, …}],
                … ,
        'deletion' :
                [ {'ID':11, 'col1':12,…}],
                …
}
```

Figure 15: Delta in JSON

```
WITH delete_1 AS (
       DELETE FROM dejima_table WHERE ID=1 AND
col1=2 AND …
), delete_2 AS (
       …
), insert_1 AS (
       INSERT INTO dejima_table (ID, col1, …) VALUES
(1, 2, …)
), insert_2 AS (
       …
) , prop_to_basetable AS (
       SELECT basetable_propagate_updates()
)

SELECT * FROM insert_1
UNION SELECT * FROM insert_2
UNION …
UNION SELECT * FROM delete_1
UNION SELECT * FROM delete_2
UNION …
UNION SELECT * FROM prop_to_basetable
```

Figure 16: Query with `WITH` clause for update propagation.

connection from the stored pair of (global transaction ID, database connection) when the global lock was acquired. From the received delta, each participant in the same Dejima group synchronize its own copy of the Dejima table and propagates the update to its own base table. We use a specific query with `WITH` clause for this purpose, as shown in Figure 16. Due to the constraints of BIRDS, we need to propagate the update to the base table immediately after making an update to the Dejima table, and we need to ensure that no other queries are executed during the execution of these queries. To this end, the `WITH` clause is used to group these queries and ensure that they are executed consecutively.

After executing the above query, if this participant shares another Dejima table, we get the delta in that Dejima table. After obtaining the delta using the aforementioned procedure, if this delta is not empty, the information is sent to the /_propagate endpoint of other participants in the same way as described above. If the participant that received the information by the /_propagate endpoint does not share any other Dejima table, or if the delta in the other Dejima table is empty, and no further update propagation occurs, an Ack is returned to the participant that invoked this endpoint. If there are other participants that share the Dejima table and are performing further update propagation, an Ack will be sent back to the participant that invoked this endpoint only when an Ack is received from all the participants in the propagation destination. If an error is detected in the database during these operations, or if the participant to which the message is being propagated replies with a Nak, the corresponding participant replies with a Nak.

A.5 Ending a Transaction

At the original participant of the transaction, if an Ack is returned from every participant that has performed the corresponding update propagation, the transaction is committed; otherwise, the transaction is aborted. This is processed using a tree two-phase commit, as shown in Figure 17. The global transaction ID and the result of

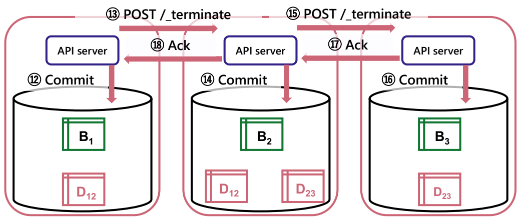


Figure 17: Tree two-phase commit.

the transaction are described in JSON format, and sent to the /_terminate endpoint of each participant involved in this transaction in a cascading manner, using the `POST` method. Upon receiving this information, each of these participants retrieves the corresponding database connection from the stored pair of (global transaction ID, database connection) when the global lock was acquired. An Ack is send back to the invoking participant in a cascading manner. Once the original participant of the transaction confirms that Ack has been returned from all the participants involved in this transaction, the transaction terminates.

B EXTENSION TO MULTIDIRECTIONAL TRANSFORMATIONS

The idea of synchronization of more than two information sources by network of bidirectional transformations are well studied (for example, by Stevens [67]) as multidirectional transformations [17]. We can consider the pair of *get* and *put* as fundamental building block for such construction, and composition patterns can be classified into

(1) sequential composition [31]

$$\begin{aligned} get(S) &= get_2(get_1(S)) \\ put(S,T) &= put_1(S, put_2(get_1(S),T)) \end{aligned}$$

that gives rise to another asymmetric lens where the output of the first *get* is fed to the second *get*;

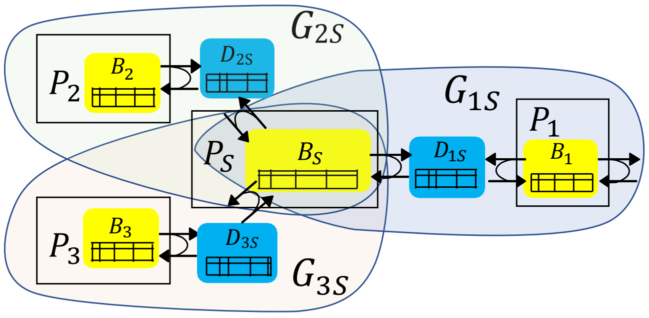


Figure 18: More expressive bidirectional view updates achieved by replacing Dejima group G_1 depicted in Figure 2 by Dejima groups G_{1S}, G_{2S} and G_{3S}, centered around a new dedicated participant P_S responsible for synchronization among original Dejima group G_1.

(2) co-targetial or co-span composition [25, 44]

$$put_{12}(S,T) = put_2(T, get_1(S))$$
$$put_{21}(S,T) = put_1(S, get_2(T))$$

that gives rise to a constraint maintainer [61] where the output of the first *get* is fed to the second *put*; and

(3) co-sourcial or span composition [25, 43]

$$put_{12}(S,C) = (get_2(C'), C')$$
$$\text{where } C' = put_1(C,S)$$
$$put_{21}(T,C) = (get_1(C'), C')$$
$$\text{where } C' = put_2(C,T)$$

that gives rise to a symmetric lens [38] where the output of the first *put* is fed to the second *get*. C represents internal state. In the above formulation, it is passed around as additional arguments but implementations could keep it entirely internal so that arguments are just S and T for each direction.

Notably, a multispan composition that connects more than two bidirectional transformations in a star shape centering around an internal state gives rise to multiary lens [28]. The same argument applies for a multicospan composition [28].

The Dejima architecture can be considered as a bipartite graph [27] where Dejima groups and base tables constitute nodes, while asymmetric bidirectional transformations constitute edges, with base tables as their sources and Dejima tables as targets. Since *get* may discard information, co-targetial composition has been considered amenable to control exposure of information ([21], for example). From this viewpoint, the transformations between two participants connected via Dejima in an identical group can be considered as co-targetial composition and thus suitable for controlling information exposures from each participant. On the other hand, this composition is strictly less expressive than co-sourcial (span) composition [28], intuitively because the span composition can store every information necessary in the internal state C to let the bidirectional transformations on both sides fully access the information. From this viewpoint, each participant can function as multiary lens [26] to support multidirectional transformation through multiple Dejimas and the internal state as its base table. Then we argue that, if we really need this expressiveness, we could replace each Dejima by another participant that is connected through each original Dejima. Figure 18 shows such case to replace Dejima D_1 in Dejima group G_1 of Figure 2 by Dejima groups G_{1S}, G_{2S} and G_{3S}, centered around a new dedicated participant P_S responsible for synchronization among original Dejima group G_1.