Ad Hoc Transactions through the Looking Glass: An Empirical Study of Application-Level Transactions in Web Applications

ZHAOGUO WANG, CHUZHE TANG, XIAODONG ZHANG, QIANMIAN YU, and BINYU ZANG, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China and Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

HAIBING GUAN, Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China

HAIBO CHEN, Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, China and Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Many transactions in web applications are constructed ad hoc in the application code. For example, developers might explicitly use locking primitives or validation procedures to coordinate critical code fragments. We refer to database operations coordinated by application code as *ad hoc transactions*. Until now, little is known about them. This paper presents the first comprehensive study on ad hoc transactions. By studying 91 ad hoc transactions among eight popular open-source web applications, we found that (i) every studied application uses ad hoc transactions (up to 16 per application), 71 of which play critical roles; (ii) compared with database transactions, concurrency control of ad hoc transactions is much more flexible; (iii) ad hoc transactions are error-prone—53 of them have correctness issues, and 33 of them are confirmed by developers; and (iv) ad hoc transactions have the potential for improving performance in contentious workloads by utilizing application semantics such as access patterns. Based on these findings, we discuss the implications of ad hoc transactions to the database research community.

CCS Concepts: • **Information systems** → **Database transaction processing**; **Web applications;**

Additional Key Words and Phrases: Ad hoc transactions

ACM Reference Format:

Zhaoguo Wang, Chuzhe Tang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2024. Ad Hoc Transactions through the Looking Glass: An Empirical Study of Application-Level Transactions in Web Applications. *ACM Trans. Datab. Syst.* 49, 1, Article 3 (February 2024), 43 pages. https://doi.org/10.1145/3638553

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62272304, 62132014, and 62172272.

Authors' addresses: Z. Wang (Corresponding author), C. Tang, X. Zhang, Q. Yu, B. Zang, and H. Chen (Corresponding author) Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, China and Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China; e-mails: {zhaoguowang, haibochen}@sjtu.edu.cn; H. Guan, Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, China.

ACM 0362-5915/2024/02-ART3
https://doi.org/10.1145/3638553

1 INTRODUCTION

Today, web applications often use database systems to persist large amounts of data, necessitating the coordination of concurrent database operations for correctness. One common approach is using database transactions. Transactions isolate concurrent database operations by encapsulating them into individual units of work. Another widely adopted approach is using the invariant validation APIs provided by **object-relational mapping (ORM)** frameworks (e.g., the validates keyword from Active Record [98]). With such APIs, developers explicitly specify invariants, such as the uniqueness of column values, in the application code and the ORM frameworks report errors on invariant violations. So far, much work has been done to investigate and improve these two approaches [8, 9, 25, 31, 39, 65, 66, 79, 97, 117, 118, 122].

However, besides these approaches, application developers are also accustomed to coordinating critical database operations ad hoc. Specifically, developers might explicitly use locking primitives and validation procedures to implement **concurrency control (CC)**, e.g., **optimistic concurrency control (OCC)**, amid the application code to coordinate critical database operations. We refer to such ad hoc coordination of database operations as *ad hoc transactions*. Developers' comments suggest that they implement ad hoc transactions for flexibility or efficiency [24].

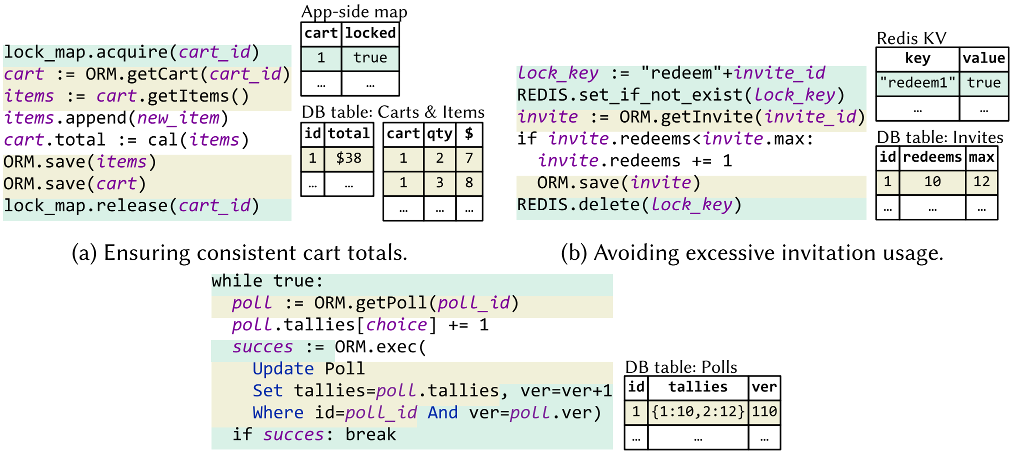
Figure 1 shows three real-world examples of ad hoc transactions from open-source web applications, Broadleaf [18], Mastodon [106], and Discourse [21]. In each example, the application code uses ORM frameworks to issue database operations and uses ad hoc constructs to coordinate them. The first two directly use locks for coordination, while the third one implements a validation-based protocol similar to OCC. As shown in the examples, ad hoc transactions are usually coupled with business logic, thus bringing difficulties to a thorough investigation. As a result, there have been few studies on ad hoc transactions. Neither their roles in web applications nor their characteristics are clearly understood.

We spent five person-years conducting a comprehensive study over 91 ad hoc transactions in eight web applications of various categories, including e-commerce, social network, forum, project management, access control, and supply chain management (Table 2). These applications are, measured by GitHub stars, the most popular ones in respective categories and developed in different languages (Java, Ruby, or Python) using different ORM frameworks (Hibernate [100], Active Record [98], and Django [30]). Our study aims to understand the characteristics of ad hoc transactions in existing web applications and their implications. Briefly, we have revealed the following interesting, alarming, and perceptive findings.

(i) Every studied application uses ad hoc transactions on critical APIs. Specifically, 71/91 ad hoc transactions are on the critical APIs in the studied web applications (Table 3). For example, there are 37 ad hoc transactions across three e-commerce applications. 31 ad hoc transactions are in critical APIs such as check-out, payment and add-cart to coordinate operations on critical data (e.g., user credits).

(ii) Ad hoc transactions' usages and implementations are much more flexible than database transactions. For example, they can perform coordination that is challenging solely with database transactions, if not impossible, such as partial coordination (22 cases), cross-request coordination (10 cases), and coordination over heterogeneous backends (8 cases). Furthermore, developers can leverage domain knowledge for optimization, such as tuning the coordination granularity to increase parallelism (14 cases) and reduce the number of locks required (58 cases).

(iii) Ad hoc transactions are prone to errors. Ad hoc transactions' flexibility comes at a cost—53 cases of ad hoc transactions manifest concurrency bugs, 28 of which even lead to severe real-world consequences, such as overcharging customers. Furthermore, server and database crashes amid ad hoc transactions can easily (90.3% of the chance) cause inconsistent states and user-perceivable buggy behaviors. While this large percentage might seem unsurprising considering the variety of



```
lock_map.acquire(cart_id)
cart := ORM.getCart(cart_id)
items := cart.getItems()
items.append(new_item)
cart.total := cal(items)
ORM.save(items)
ORM.save(cart)
lock_map.release(cart_id)
```

App-side map

cart	locked
1	true
…	…

DB table: Carts & Items

id	total
1	$38
…	…

cart	qty	$
1	2	7
1	3	8
…	…	…

(a) Ensuring consistent cart totals.

```
lock_key := "redeem"+invite_id
REDIS.set_if_not_exist(lock_key)
invite := ORM.getInvite(invite_id)
if invite.redeems<invite.max:
  invite.redeems += 1
  ORM.save(invite)
REDIS.delete(lock_key)
```

Redis KV

key	value
"redeem1"	true
…	…

DB table: Invites

id	redeems	max
1	10	12
…	…	…

(b) Avoiding excessive invitation usage.

```
while true:
  poll := ORM.getPoll(poll_id)
  poll.tallies[choice] += 1
  succes := ORM.exec(
    Update Poll
    Set tallies=poll.tallies, ver=ver+1
    Where id=poll_id And ver=poll.ver)
  if succes: break
```

DB table: Polls

id	tallies	ver
1	{1:10,2:12}	110
…	…	…

(c) Ensuring accurate poll statistics.

Fig. 1. Ad hoc transaction examples. Coordinated database accesses are shaded yellow; ad hoc constructs are shaded green.

ad hoc transaction implementations, our study is the first to provide a detailed analysis of this phenomenon. For example, we find that 11 cases have more than one issue, requiring independent fixes. Among all issues, incorrect primitive implementations, such as locks, are the most common cause (47 cases). We have submitted 20 issue reports (covering 46 cases) to developer communities; 7 of them (covering 33 cases) have been acknowledged.

(iv) Ad hoc transactions can have performance benefits under high-contention workloads. Using application semantics such as access patterns, ad hoc transactions' CC could be implemented in a simple yet precise way. Thus, they can avoid false conflicts under high contention workloads. For example, an ad hoc transaction may leverage the knowledge of accessed columns to use column-level locks for coordination, which can achieve up to 1.3× API performance improvement compared to row-level locking by avoiding false conflicts on the contented rows.

The prevalence of ad hoc transactions and their unique characteristics suggest the potential of improving existing database systems that support these applications. Finally, we discuss the implications of our findings on future database and storage systems research.

2 BACKGROUND AND MOTIVATION

2.1 Concurrency Control in Web Applications

Today, web applications often use standalone **relational database management system (RDBMS)** to manage and persist data so that developers can focus on writing business logic. As web applications are prominently written in object-oriented languages, most applications manipulate relational data with the help of ORM frameworks such as Hibernate [100] and Active Record [98]. These frameworks can transparently generate SQL statements that fetch and persist data according to the application code. Fetched relational data is presented as in-memory, application runtime objects, which we refer to as *ORM-mapped objects*. Furthermore, ORMs also provide interfaces to assist developers in coordinating concurrent database accesses: *database transaction APIs* and *invariant validation APIs*.

ORM frameworks usually allow developers to use database transactions explicitly, with interfaces that directly translate to Transaction Start, Commit, and Abort statements. Developers

Table 1. Comparison with *Feral Concurrency Control* [9] and *ACIDRain* [119]

Study	Subject	Focus	Issue types
Feral CC	ORMs' invariant validation APIs.	Characteristics and correctness.	Insufficient isolation.
ACIDRain	Database transactions.	Correctness.	Insufficient isolation and incorrect transaction scopes.
This work	Ad hoc transactions.	Characteristics, correctness, and performance.	Incorrect synchronization primitives, transaction scopes, and failure handling.

use them to encapsulate multiple database operations into units of work, and the database system takes the responsibility of coordination. Furthermore, ORM frameworks also allow developers to configure the isolation level for specific transactions. However, most web applications use the default isolation level of the database system [119].

Besides database transactions, ORMs also provide built-in invariant validation APIs. For example, Active Record [98] provides validation and association keywords, such as validates and belongs_to. Developers use them to explicitly specify invariants, such as the uniqueness of column values and the presence of associated rows, in the application code. Active Record checks invariants upon database writes and report errors on violations. Checks are typically done by examining the to-be-persisted ORM-mapped objects and related rows fetched from the database systems.

Invariant validation differs from database transactions. The latter coordinates every *database operation* according to given isolation requirements; the former handles concurrency by directly examining *database states* to prevent the specified invalid outcomes only.

2.2 Existing Studies on Concurrency Control in Web Applications

Researchers have studied how database-backed web applications handle concurrency (Table 1). The major difference between these works and ours lies in the coordination approach being studied. Bailis et al. [9] studied "feral" CC—ORM's invariant validation APIs, and Warszawski and Bailis [119] studied database transactions, while this work targets a third, much less modular approach, ad hoc transactions. Consequently, we examine different aspects and have arrived at new and interesting findings.

Specifically, Bailis et al. studied how Rails [99] applications adopt invariant validation APIs to handle concurrency, and they analyzed the soundness of this approach. They have found that application-level invariant validations are used much more often than database transactions. Furthermore, with the theory of invariant confluence [8], they have found that the majority of the validations are sound, i.e., they preserve invariants even under concurrent execution using weak isolation levels such as Read Committed, while the remainders do not. Meanwhile, Warszawski and Bailis focused on the correctness of database transaction usages in web e-commerce applications. They analyzed SQL logs to identify non-serial API executions that potentially violate application invariants. By manually checking potential violations, they have identified 22 bugs caused by *insufficient isolation levels* and *incorrect transaction scopes*.

In contrast, we examine the characteristics (Section 3), correctness (Section 4), and performance (Section 5) of ad hoc transactions. We believe our results complement those of Bailis et al. in understanding application-level CC and may benefit Warszawski and Bailis's method as ad hoc transactions are composed of application-level constructs, which cannot be captured by SQL logs and thus cause false conflicts for their method [119, Section 3.2].

Table 2. The Applications Corpus

Application	Category	Language/ORM	RDBMS	Stars	Contri-butors
Discourse [21]	Forum	Ruby/Active Record	PG	33.8k	776
Mastodon [106]	Social network	Ruby/Active Record	PG	24.6k	644
Spree [113]	E-commerce	Ruby/Active Record	PG, MY	11.4k	855
Redmine [62]	Project mgmt.	Ruby/Active Record	PG, MY, +	4.2k	8
Broadleaf [18]	E-commerce	Java/Hibernate	PG, MY, +	1.5k	73
SCM Suite [33]	Supply chain mgmt.	Java/Hibernate	PG, MY	1.5k	2
JumpServer [37]	Access control	Python/Django	PG, MY, +	16.8k	88
Saleor [108]	E-commerce	Python/Django	PG, MY, +	13.9k	181

The “RDBMS” column lists supported RDBMSs and “PG/MY/+” refers to PostgreSQL/MySQL/others.

2.3 Ad Hoc Transactions in the Wild

Besides database transactions and ORM-provided invariant validation, we have observed a third CC approach in web applications—ad hoc transactions. As shown in Figure 1, like database transactions, ad hoc transactions provide isolation semantics such as serializability to database operations. The difference is that ad hoc transactions coordinate operations with application code—it is the application developers, instead of database developers, who design and implement the CC. More concretely, ad hoc transactions are pieces of application code that access shared states to perform business logic while also being responsible for coordinating their own executions. Such coordination includes deciding whether or not to execute certain operations and when to execute them. For example, in Figure 1(a), API handlers share the lock_map object for determining when to execute the database operations for adding an item to a user’s cart, and in Figure 1(c), the ver field determines whether or not the poll result update can be performed. Both ORM’s invariant validation APIs and ad hoc transactions operate at the application level. However, the difference lies in how they ensure correctness. The former looks at database states for invariant violation; the latter directly isolates concurrent database operations. For example, Figures 1(a) and 1(b) use locks to isolate conflicting operations, e.g., the concurrent reading and writing of the same cart. Similarly, Figure 1(c) uses version checks to detect conflicting changes and ensure **read–modify–writes (RMWs)** are atomic. In contrast, with ORM’s invariant validation, these conflicting accesses can freely interleave; application invariants, such as the non-negativity of total fields, are checked only when data is written back to the RDBMS.

To understand ad hoc transactions’ roles and criticality in web applications, we investigated eight representative applications of six categories (Table 2). They are the most popular web applications in each category[1] and developed in different languages with different ORM frameworks. For example, Broadleaf [18] is the highest star-ed Java e-commerce application on GitHub and Spree [113] is the most popular e-commerce application in Ruby. We build the application corpus by first including web applications that we already have a basic understanding of, which include Broadleaf, Spree, and SCM Suite. We then expanded the corpus with top stared applications listed on GitHub that use the language-ORM combinations mentioned above. In the second phase, we skipped applications that fell into the same category as those already present in the corpus (diaspora* [29]), applications that were not actively maintained at the time (Refinery CMS [104]

[1] Redmine [62] is the second popular project management application now. Its popularity has waned since we picked it as the investigation target.

Table 3. Ad Hoc Transactions are Mainly used in Core APIs

Application	Core APIs using ad hoc transactions	Cases
Discourse	Posting, image upload, and notification.	8/13
Mastodon	Posting, polls, messaging, and viewing.	10/16
Spree	Check-out and cart modification.	10/10
Redmine	Issue tracking, metadata management, and attachments.	6/9
Broadleaf	Check-out and cart modification.	6/11
SCM Suite	Account management and merchandise information tracking.	11/11
JumpServer	Granting privileges and asset updates.	5/5
Saleor	Check-out, payment, refund, and stock management.	15/16

and Huginn [57]), and applications that were too complex for us to understand fully (GitLab [41]). It should be noted that there are applications with more stars in categories not covered here (e.g., blogging). Nevertheless, we believe our selection is sufficiently representative to yield findings of reasonable generality. To locate ad hoc transactions, we first search the keywords such as "lock," "concurrency," and "consistency" in the codebase, the commit histories, and the issue trackers. Then, we manually identify coordination code that isolates database operations and the purpose of those operations.

> FINDING 1. *Every studied application uses ad hoc transactions. Among the 91 ad hoc transactions in total, 71 cases are considered critical to the web applications.*

Table 3 shows the study result on ad hoc transactions' criticality. For the e-commerce applications, we consider an ad hoc transaction critical if it resides in their core APIs such as check-out and add-cart to ensure safe shopping. For example, an ad hoc transaction may coordinate the reading and writing coupon data to avoid coupon overuse. Among the three popular e-commerce applications, Broadleaf [18], Spree [113], and Saleor [108], there are 37 ad hoc transactions in total, and 31 of them are critical. Specifically, 13 cases ensure that orders are accepted only when the stock quantity is sufficient, and 5 avoid inconsistent capture of payment. Interestingly, all these applications have ad hoc transactions to ensure sufficient stock quantity and coupon validity. Core APIs of other applications are listed in Table 3.

Considering their importance in web applications, we further investigate ad hoc transactions to answer the following questions.

- How are ad hoc transactions constructed among applications code? (Section 3)
- Do ad hoc transactions always work correctly? (Section 4)
- How is ad hoc transactions' performance, especially in comparison with database transactions (Section 5)?

3 CHARACTERISTICS OF AD HOC TRANSACTIONS

We have carefully studied the 91 identified ad hoc transaction cases. As shown in Table 4, an interesting but not surprising finding is that, even though developers implement ad hoc transactions in various ways, these cases can still be classified into *pessimistic* ad hoc transactions (65/91) and *optimistic* ad hoc transactions (26/91). In pessimistic cases, developers explicitly use locks to block conflicting database operations in ad hoc transactions. This method is similar to **two-phase locking**

Table 4. Statistics of Identified Ad Hoc Transitions

	Cases		CC style	
Application	**Total**	**Buggy**	**Pes.**	**Opt.**
Discourse	13	13	10	3
Mastodon	16	11	11	5
Spree	10	10	4	6
Redmine	9	1	6	3
Broadleaf	11	7	5	6
SCM Suite	11†	8	8	3
JumpServer	5	0	5	0
Saleor	16	3	16	0
Total	91	53	65	26

"Pes." and "opt." refer to *pessimistic* and *optimistic* concurrency control style, respectively. †SCM Suite generates source code for different suppliers from templates; only cases in templates are counted. In its (generated) demo, there are 164+3 cases.

(2PL) and its variants commonly used by existing database systems [36, 40, 43, 58, 64, 75, 77, 102]. Unlike database transactions, pessimistic ad hoc transactions' locking primitives are usually implemented from scratch by application developers (e.g., Figures 1(a) and 1(b)) or provided by other systems (see Section 3.2). Meanwhile, optimistic ad hoc transactions execute operations aggressively and validate the execution result before writing updates back to the database system (Figure 1(c)). This approach is similar to OCC and its variants used in existing database systems [55, 56, 61, 79, 101, 117].

Though ad hoc transactions can be straightforwardly categorized as either pessimistic or optimistic, they are nonetheless notably different in terms of usages and implementations. Specifically, (i) how do ad hoc transactions blend in with and coordinate business logic? (ii) how is their CC designed and implemented? (iii) what are their coordination granularities? (iv) how do they handle failures?

With these questions in mind, we examine ad hoc transactions and compare them with database transactions to gain further insights. For the comparison, we considered database transactions from MySQL 8.0.25 and PostgreSQL 13.5, the two most popular open-source RDBMSs [26] compatible with the applications (Table 2).

3.1 What do Ad Hoc Transactions Coordinate?

In writing ad hoc transactions, developers explicitly place ad hoc coordination constructs among the business logic. This approach gives them the flexibility of choosing which and how operations are coordinated, enabling partial coordination, cross–HTTP request coordination, and coordination with non-database operations.

Finding 2. *Among the 91 ad hoc transactions studied, 22 only coordinate a portion of database operations in their scopes, and 10 coordinate operations across multiple requests. Besides, 8 cases coordinate database operations along with non-database operations.*

```
 1 IN: sku_id, requested
 2 lock(sku_id)
 3 sku := SELECT * FROM SKUs WHERE id=sku_id
 4 if sku.quantity >= requested:
 5    sku.quantity -= requested
 6    // The followig statements are auto-generated by ORM.save(sku).
 7    TRANSACTION START
 8    UPDATE SKUs SET quantity=sku.quantity WHERE id=sku.id
 9    UPDATE Products SET updated_at=now() WHERE id=sku.product_id
10    category_ids := SELECT category_id
11                       FROM Categories JOIN ProductCategories USING category_id
12                       WHERE product_id=sku.product_id
13    UPDATE Categories SET updated_at=now() WHERE id IN category_ids
14    TRANSACTION COMMIT
15 unlock(sku_id)
```

Fig. 2. Spree uses an ad hoc transaction to coordinate specific database operations when processing customer orders.

3.1.1 All Database Operations vs. Specific Database Operations. As ad hoc transactions' coordination is explicitly written by application developers, developers can coordinate only specific database operations instead of all operations in the transaction scope. Consider the example from the Spree e-commerce application [113] shown in Figure 2. This transaction processes customer orders. It first fetches the **stock-keeping unit (SKU)** data from the SKUs table, checks and updates the SKU's stock quantity, then persists changes to the database system by invoking the ORM.save() method. ORM.save() automatically starts a database transaction, within which it issues three updates and one query (line 8–13). This transaction is running in the RDBMS' default isolation level.[2] The first update changes the quantity in the SKUs table, and other updates refresh the update_at timestamps of corresponding Products and Categories rows. Categories rows are identified by querying the ProductCategories table, which encodes the many-to-many relationship between products and categories. In this example, the only critical operations are those over SKUs (lines 3 and 8). Therefore, developers explicitly lock over sku_id in their ad hoc transaction implementation. Other operations such as product and category updates (lines 9 and 13) require no coordination but are still in the lock scope, as the application-level ORM.save() call automatically generates them.

In this example, replacing the lock()/unlock() primitives with explicit Transaction Start/Commit may worsen performance, as all the updates will be performed under the same isolation level. With explicit transactions, ORM.save() would not initiate a nested transaction or create a savepoint. Instead, the explicit outer transaction would execute its query and update operations. We first consider the performance issue with MySQL, one of Spree's supported RDBMSs (Table 2). Serializable isolation must be used since all MySQL's non-Serializable isolation levels will introduce lost updates due to the RMW operations over SKUs [73, Section 7.3.3.3]. Unfortunately, two Serializable transactions would deadlock when they attempt upgrading to writer locks at line 13 after acquiring reader locks on the same Categories row at line 10. However, with ad hoc transactions, only the critical SKU operations are serialized, and Categories accesses are executed in MySQL's default isolation level, Repeatable Read, which does not acquire reader locks [83, Section 15.7.2.3].

[2] MySQL defaults to Repeatable Read; PostgreSQL defaults to Read Committed.

```
Request 1 // Fetch a post & increment view count.
  in: post_id
  post := Select * From Posts Where id=post_id
  Update Post Set view_cnt=view_cnt+1, ver=ver+1 Where id=post.id
  response render(post) // This response includes the version.
Request 2: // Detect interruptions & apply user updates.
  in: post_id, new_content, prev_ver
  lock(post_id)
  current := Select * From Posts Where id=post_id
  if current.ver!=prev_ver:
    unlock(post_id)
    response FAILURE
  Update Posts Set content=new_content, ver=ver+1 Where id=post_id
  unlock(post_id)
  response SUCCESS
```

Fig. 3. Discourse uses an ad hoc transaction to coordinate the post editing process that spans multiple requests.

Besides MySQL, other database systems might also have similar issues. Consider using PostgreSQL to back Spree, where Repeatable Read is the weakest available isolation level that avoids lost updates on SKUs in this example. PostgreSQL implements Repeatable Read as an alias for Snapshot Isolation. When concurrent transactions update different SKUs but the same Categories row and cause write–write conflicts, PostgreSQL will abort transactions according to Snapshot Isolation's *first-committer-wins* property [15]. In contrast, ad hoc transactions' ORM-generated Categories accesses are executed under PostgreSQL's default isolation level, Repeatable Read, where conflict writes will not cause aborts [49, Section 13.2.2].

Ideally, developers should exclude these timestamp updates from the scope of database transactions or switch the isolation level with database interfaces [72]. However, neither could be applied to the above example, as the ORM hides the generation of such database operations. 22 ad hoc transactions coordinate only a portion of the database operations in the transaction scope. The other operations require no coordination but are located in the transaction scope as they are either automatically generated by the ORM or needed by critical operations. However, it is difficult for the database transaction to provide such flexibility.

3.1.2 Individual Requests vs. Multiple Requests. It is a performance anti-pattern for database transactions to span multiple HTTP requests, introducing **long-lived transactions (LLTs)**. However, 10 ad hoc transactions coordinate database operations across multiple requests. Figure 3 shows an example derived from the Discourse forum application [21] of editing a post that spans two user requests. The user fetches the post content for local editing in the first request. Then, the user's edits are applied in the second request. This ad hoc transaction ensures that other concurrent edits do not overwrite the content read by the first request when editing the post. Specifically, developers use an optimistic ad hoc transaction to ensure the consistency of the post content. They associate a version with each post to track updates. Before updating a post, the ad hoc transaction checks the consistency (i.e., not overwritten) by validating the version. Furthermore, it needs to use a lock to ensure the validate-and-commit atomicity. If the validation fails, the current request handler will not update the content, thus avoiding overwriting others' changes. However, the view count increment in the previous request handler cannot be rolled back. Normally, web applications choose optimistic coordination instead of pessimistic coordination to coordinate multiple requests to avoid long blocking.

```
Create Post
  in: follower_id, post_id, content
  lock(post_id)
  Insert Into Posts Value (post_id, content)
  REDIS.add_to_set("timeline"+follower_id, post_id)
  unlock(post_id)
Delete Post
  in: follower_id, post_id
  lock(post_id)
  REDIS.delete_from_set("timeline"+follower_id, post_id)
  Delete From Posts Where id=post_id
  unlock(post_id)
```

Fig. 4. Mastodon uses ad hoc transactions to coordinate both database operations and access to Redis key–value store.

Extensions to database transactions were proposed for LLTs, such as *Sagas* [39] and *savepoints* [44, 76]. They usually provide (potentially unnecessarily) stronger semantics than what ad hoc transactions provide here. To use Sagas, developers have to decompose an LLT into subtransactions accompanied with compensation transactions. When any subtransaction aborts, compensation transactions of prior-committed subtransactions will be invoked, negating their effects as if the LLT has never been executed. This semantic is different from the ad hoc transaction across multiple requests. The above ad hoc transaction only aborts the request handler that detects conflicts. Alternatively, developers can set savepoints after handling each request when coordinating multi-request user interactions with conventional, long-lived database transactions. When the application detects an error (except for fatal errors such as deadlocks), it can explicitly roll back the transaction state to previously set savepoints instead of aborting the entire LLT. However, in some RDBMSs such as MySQL, LLTs block all other conflicting transactions until it commits, i.e., finishing the last request. For the above example, concurrent transactions which update view_cnt in the Posts table will be unnecessarily blocked. Furthermore, data written by previous requests in LLT could be lost if the application server fails midway.

3.1.3 *Database Operations vs. Non-Database Operations.*

The flexibility of ad hoc transactions is also reflected in coordinating non-database operations. A web application may use several storage systems to persist its data. Thus, it needs to ensure data consistency across different systems. There are eight cases of ad hoc transactions that coordinate both database operations and non-database operations, such as operations over in-memory shared variables, local file systems, and remote object/**key–value (KV)** stores. Consider the example shown in Figure 4, which is simplified from the timeline feature of the Mastodon social network application [106]. It uses a Redis KV store and an RDBMS as its backend storage. Redis holds the IDs of posts shown on each user's timeline, while the concrete post contents are resident in the RDBMS. To ensure correctness, Mastodon must guarantee the consistency between the post contents in the RDBMS and the post IDs in Redis. Specifically, the post IDs in Redis should always refer to post contents in the RDBMS, which cannot be achieved solely with database transactions. Thus, developers implement ad hoc transactions to coordinate these operations. Note that only the post is locked in this example because the operations over Redis timelines commute.

In general, when the business logic requires data from multiple storage systems (including multiple RDBMSs) to stay consistent, the alternative option is to use distributed transactions, such as WS-Atomic Transaction [80, 81] or XA transactions [124]. However, storage systems rarely support

```
IN: item_id
Transaction Start
alloc := Select * From Allocations Where item_id=item_id For Update
stock := Select * From Stocks Where id=alloc.stock_id For Update
if alloc.qty > stock.qty: Transaction Abort
else:
    Update Allocations Set qty=0 Where id=alloc.id
    Update Stocks Set qty=qty-alloc.qty Where id=stock.id
    Transaction Commit
```

Fig. 5. Saleor uses Select For Update statements for synchronization in an ad hoc transaction that coordinates stock allocation.

such distributed transaction protocols, which necessitate ad hoc transactions. Dey et al. [27, 28] designed a protocol, Cherry Garcia, providing ACID transactions over multiple KV stores at the application level. In addition to a KV interface, it poses further requirements on KV stores, such as the ability to set user-defined metadata. Therefore, Cherry Garcia cannot directly replace ad hoc transactions since other accessed storage systems do not necessarily meet these requirements.

3.2 How Is Their Coordination Implemented?

Developers need to manually coordinate ad hoc transactions, including locking (for pessimistic cases) and validation (for optimistic cases). However, the locking primitives and validation procedures usually have different implementations.

Finding 3. *There are 7 different lock implementations and 2 validation implementations among the 8 applications we studied. Except for Broadleaf, developers consistently use the same lock/validation implementation in individual applications.*

3.2.1 Existing Systems' Locks vs. Hand-Crafted Locks. All eight studied applications have lock-based pessimistic ad hoc transactions. They usually use a single locking primitive implementation, provided by either existing systems or developers themselves.

Four applications directly use the locking primitives provided by the database systems or languages runtimes. Specifically, Spree [113], Saleor [108], and Redmine [62] use the database Select For Update statements, while SCM Suite [33] implements ad hoc transactions based on the Java synchronized keyword. Most commercial databases accept Select For Update statements, which atomically fetch target rows and acquire corresponding writer locks. The lock will be released when the currently active transaction ends. The example in Figure 5 is simplified from the Saleor e-commerce application [108], where developers acquire database locks on the stock and the stock allocation with Select For Update. The lock is released after the stock's sufficiency is checked and the allocation is applied. Thus, ad hoc transactions must enclose the critical section in a database transaction to use the database locks. However, this database transaction could be configured with a weak isolation level such as Read Committed.

Three other applications, Discourse [21], Mastodon [106], and JumpServer [37], have locks implemented from scratch. Interestingly, they all store lock information, including lock keys and status (locked/unlocked), in the Redis KV store. However, as shown in Figure 6, their implementation details are different. Mastodon developers use the Redis SETNX (short for SET if Not eXists) command to insert an entry for the requested lock (Figure 6(a)). Similar to the **Compare and Swap (CAS)** instruction, this command succeeds only if no entry with the same key exists or is

```
Lock
  IN: key
  token := rand()
  while time spent < 10s:
    if REDIS.SETNX(key=key, value=token, expire=10s):
      // SETNX succeeds if the key is absent/expired.
      return TRUE, token
    sleep 0.1s
  return FALSE
```

```
Unlock
  IN: key, token
  REDIS.EVAL do
    // EVAL runs the given script atomically.
    if REDIS.GET(key) == token:
      REDIS.DEL(key)
      return TRUE
    else:
      return FALSE
  end
```

(a) The Redis lock used in Mastodon that relies on the SETNX and EVAL Redis commands for conditional put/delete.

```
Lock
  IN: key
  while TRUE:
    now := REDIS.TIME()
    expire := now + 1m
    REDIS.UNWATCH() // Unset the watch set.
    REDIS.WATCH(key)  // Add key to the watch set.
    prev_expire := REDIS.GET(key)
    if prev_expire != NULL and prev_expire >= now:
      REDIS.UNWATCH()
      continue
    REDIS.MULTI() // Start queuing commands.
    REDIS.SET(key=key, value=expire)
    REDIS.EXPIREAT(key=key, expire=expire+1s)
    if REDIS.EXEC(): // Run queued commands atomically
      return expire   // if the watch set is unchanged.
    sleep 1ms
```

```
Unlock
  IN: key, expire
  now := REDIS.TIME()
  if now <= expire:
    REDIS.UNWATCH()
    REDIS.WATCH(key)
    current_expire := REDIS.GET(key)
    if current_expire == expire:
      REDIS.MULTI()
      REDIS.DEL(key)
      REDIS.EXEC()
    else:
      REDIS.UNWATCH()
  else:
    warn("lock held too long")
```

(b) The Redis lock used in Discourse that relies on the WATCH/UNWATCH, MULTI and EXEC Redis commands to perform optimistic concurrency control.

Fig. 6. Two flavors of Redis-based lock implementations are found among Mastodon, Discourse, and JumpServer.

not expired. Since the lock entry is written with an expiration time, other threads might overwrite it once it has expired. To avoid accidental release of locks acquired by other threads, during unlock, a thread needs to atomically check whether the current lock entry has been overwritten and delete it only if it remains unchanged. Mastodon developers generate random tokens at lock time to distinguish threads that write to the same lock entry key. The atomicity of check and delete is achieved by a Redis EVAL command which accepts a Lua script and executes it with other Redis activities paused. In contrast, Discourse developers use a combination of WATCH, MULTI, and EXEC commands to optimistically ensure the atomicity of checking existing locks and setting new locks (Figure 6(b)). The MULTI command instructs Redis to start queuing subsequent commands instead of executing them immediately, and the EXEC command conditionally executes the queued commands in an atomic manner. Discourse developers use these commands to make writing a new lock entry and its expiration time atomic.[3] The EXEC command only succeeds when the keys marked by a prior WATCH command remain unchanged. As the lock key is first watched and then

[3] The expiration time can be set along the first queued SET command as an extra argument. It is unclear why Discourse developers chose to issue two commands instead. Still, using MULTI and EXEC is required to handle concurrent locking.

```
Lock
  IN: order, run_id // run_id is an UUID generated each time Broadleaf starts up.
  Transaction Start
  count := Select count(*) From OrderLocks Where order_id=order.id And run_id=run_id
  if count == 0:
    Insert Into OrderLocks (order_id, run_id, locked) Value (order.id, run_id, 'Y')
  else:
    row_changed := Update OrderLocks Set locked='Y'
                         Where order_id=order.id And run_id=run_id And locked='N'
    if row_changed != 1:
      Transaction Abort
      return FALSE
  Transaction Commit
  return TRUE
Unlock
  IN: order, run_id
  Update OrderLocks Set locked='N' Where order_id=order.id And run_id=run_id
```

(a) The database-based order lock used in Broadleaf.

```
Lock
  IN: session_id // session_id identifies unique user session.
  lock = SESSION_LOCKS.get(session_id) // SESSION_LOCKS is a global concurrent map.
  if lock == NULL:
    synchronized(SESSION_LOCKS) do // synchronized serializes concurrent lock creation.
      lock = SESSION_LOCKS.get(session_id)
      if lock == NULL:
        lock := new ReentrantLock()
        if SESSION_LOCKS.is_full():
          SESSION_LOCKS.evict_one_by_lru()
        SESSION_LOCKS.put(session_id, lock)
    end
  lock.lock()
  return lock
Unlock
  IN: acquired_lock
  acquired_lock.unlock()
```

(b) The server-side in-memory session lock (lease) used in Broadleaf.

Fig. 7. Broadleaf developers implemented various locking primitives.

read using the GET command, any concurrent change to the lock entry will be detected when EXEC runs, and no queued command will be executed on conflicts. Unlike Mastodon's lock implementation, which resembles a single-object CAS instruction, Discourse's mechanism is more of an OCC protocol. As a result, Discourse's Redis lock requires six additional round trips compared to Mastodon's, which only needs one [91]. JumpServer uses SETNX to implement locks as Mastodon; it also adds a re-entrant feature, allowing locks to be acquired by the same thread multiple times.

Broadleaf [18] is the only application using both home-grown lock implementations and existing systems' primitives—the Java synchronized keyword. More interestingly, it has three home-grown implementations. The first one uses a dedicated database table to store lock entries as individual rows, similar to the Redis-based locks mentioned earlier (see Figure 7(a)). It ensures atomicity between checking the lock status and updating the lock entry by using a database transaction. Furthermore, as database systems do not silently "expire" (i.e., delete) a row, the unlock procedure is a simple Update statement that releases the corresponding lock entry. The other two implementations use in-memory maps for lock information and differ in the specific maps used. One directly uses a concurrent map from the standard library, ConcurrentHashMap, and the other uses a customized ConcurrentHashMap where developers added a **least recently used (LRU)** eviction policy to remove excessive lock entries. Figure 7(b) shows the procedures used for in-memory session locks, which are built on the LRU-enabled concurrent map. Interestingly, although SESSION_LOCKS is a concurrent data structure, developers still have to use the synchronized keyword to prevent threads from overwriting each others' lock entry since the put method always succeeds regardless of whether the lock key is present. We find no clear evidence that these different implementations serve different purposes. However, we do find that different developers have introduced these implementations.

3.2.2 ORM-Assisted Validation vs. Hand-Crafted Validation. 6 out of 8 studied applications have validation-based optimistic ad hoc transactions. Their validation procedures are either provided by the ORM framework or developers themselves.

There are 4 applications that use ORM-provided validation procedures via framework-specific interfaces. For example, Active Record recognizes columns named lock_version and uses them to store versions for individual rows. Upon each update, as shown in Figure 1(c), Active Record automatically adds version checking to the Where clause and increment version along with user-initiated updates, ensuring the atomicity of validation and commit.

When using hand-crafted validation procedures, developers must ensure the atomicity of validation and commit. As shown in the listing from Section 3.1.2, additional locks are employed for this purpose. All validation procedures in Discourse's and SCM Suite's optimistic ad hoc transactions are manually implemented. Broadleaf uses both implementations, introduced by different developers.

Remarks. Primitive implementations vary across different applications and even in the same application. However, we did not find any obvious reason for developers preferring one particular implementation over others. We relate different implementations with different correctness issues in Section 4 and also compare their performance in Section 5.

3.3 What are Their Coordination Granularities?

Developers often have a deep understanding of applications that enables them to customize the coordination granularity. Intuitively, one might think of *finer-grained coordination* than database transactions. For example, an ad hoc transaction can coordinate at the column level and only focus on the accesses to specific columns since developers have the precise knowledge of which columns are needed by the business logic. This can reduce false conflicts caused by row-based coordination [46]. However, ad hoc transactions also employ *coarser-grained coordination* than database transactions. Specifically, ad hoc transactions often group multiple accesses together and coordinate them with a single lock. This can largely reduce ad hoc transactions' CC complexity and avoid deadlocks.

> FINDING 4. *Among the 91 studied ad hoc transactions, 14 cases perform fine-grained coordination such as column-based coordination, while 58 cases perform coarse-grained operations, i.e., using a single lock to coordinate multiple operations. 9 cases implement both types of coordination for different accesses.*

3.3.1 Single Access vs. Multiple Accesses. A lock in ad hoc transactions can coordinate arbitrary database accesses. According to our study, 58 ad hoc transactions use only one lock to coordinate multiple database accesses. This is possible as developers can identify and utilize two distinct access patterns.

Associated Accesses. Given two database rows, r_1 and r_2, if accesses to r_2 always happen in a transaction that also accesses r_1, we say r_2 is associatively accessed with r_1 and refer to this access pattern as the *associated access* pattern. Access to rows associated with a one-to-many relationship, such as an is-part-of relationship, often follows this pattern. Consider the example in Broadleaf [18], shown in Figure 1(a). A cart is represented as one Carts row and several Items rows. When a user modifies the cart, the transaction will associatively access these rows. The associated access pattern provides an opportunity of replacing multiple locks (e.g., row locks) with one lock that coordinates these accesses. In the above example, developers use a single cart lock to coordinate accesses to both tables, Carts and Items. This lock explicitly serializes conflicting transactions up front, thus avoiding potential aborts when using database transactions. In PostgreSQL, the Carts update in one transaction aborts all conflicting transactions that happen before the update due to write–write conflicts. In MySQL, both the Carts update and the Items insert can form deadlocks, as both tables might be locked in shared mode by other transactions.

There are about 37 ad hoc transactions that leverage the associated access pattern. For all the cases we studied, the associated rows are connected by either one-to-many or one-to-one relationships. We find that these one-to-many relationships stem from the application-specific data modeling that reflects the business semantics, such as the relationship between carts and items in the above example. Meanwhile, these one-to-one relationships come from inheritance. For example, Broadleaf uses a Bundled_Items table to store data for items that represent sale bundles. When querying one bundle item, two database operations are issued to the Items and Bundled_Items tables. It should be noted that inheritance can be implemented differently and does not necessarily introduce associated accesses, e.g., by merging both Items and Bundled_Items tables into one monolithic table [74, Section 2.11].

Read–Modify–Writes (RMWs). RMW means that a transaction first queries the data from the database system, then makes a modification accordingly, and finally persist the modification back to the database system. In a 2PL system without sufficient deadlock prevention mechanisms, such as MySQL, there can be a deadlock if two concurrent transactions perform the RMW on the same row. Assuming both transactions use Serializable isolation, if they both have successfully acquired reader locks, then their updates block each other, causing deadlocks. Note that MySQL's non-Serializable isolation levels does not prevent lost updates [73, Section 7.3.3.3], which necessitate the use of Serializable. Consider the example shown in Figure 1(b), in the forum application Discourse [21], RMW operations are issued when creating a new account via invitations. The invitation is first read from the database system. After checking its validity, it gets updated and written back to the database system. If two users concurrently use one invitation to join the forum, a deadlock can easily appear, making both users unable to succeed.

To mitigate this, developers craft ad hoc transactions to acquire exclusive locks before the first reads, avoiding possible deadlocks. 56 out of 91 cases leverage the RMW access pattern. Among them, 35 cases also utilize the associated access pattern.

Discussion. Reducing the number of locks simplifies the implementation and avoids potential deadlocks. However, such optimizations can rarely be used in database systems because they highly rely on application semantics. One might think of using static analysis to identify those special patterns. But this is not trivial, especially for detecting the associated access pattern. This

```
1  Create Post
2    IN: topic_id, content
3    lock("create_post"+topic_id)
4    next_post_id := Select max_post From Topics Where id=topic_id
5    Insert Into Posts Value (next_post_id, content, topic_id)
6    Update Topics Set max_post=max_post+1 Where id=topic_id
7    unlock("create_post"+topic_id)
8  Toggle Answer
9    IN: topic_id, post_id
10   lock("toggle_answer"+topic_id)
11   Update Posts Set is_answer=true Where id=post_id
12   Update Topics Set answer=post_id Where id=topic_id
13   unlock("toggle_answer"+topic_id)
```

Fig. 8. Discourse uses ad hoc transactions to coordinate at the column granularity to avoid row-level conflicts between the create-post and toggle-answer APIs.

```
IN: post_id, new_content, old_content
lock(post_id)
current := Select * From Posts Where id=post_id
if current.content!=old_content:
    unlock(post_id)
    response FAILURE
Update Posts Set content=new_content Where id=post_id
unlock(post_id)
response SUCCESS
```

Fig. 9. Discourse uses optimistic ad hoc transactions to coordinate concurrent updates to the content column.

is because one needs to analyze every line of code to ensure those accesses are always together, and web applications usually have a large codebase. For instance, our studied application has 160.4 k lines of code on average. Besides, most applications use ORMs to hide the database access details, making the analysis more challenging.

3.3.2 Fine-Grained vs. Coarse-Grained. Coordinating at a finer granularity than existing database systems has an obvious advantage in avoiding false conflicts. We find ad hoc transactions' fine-grained coordination are either based on columns or predicates.

Column-Based vs. Row-Based. Fields of ORM-mapped objects correspond to database columns. Developers could coordinate database accesses at the column granularity if they know which fields are used. For example, in the forum application Discourse [21], two transactions, create-post and toggle-answer, will issue the following database operations accessing the Topics table (Figure 8). line 6 increments the max_post field; line 12 sets the answer field. Though these operations have no column-level conflicts, if they access the same row, an database system using row locks cannot execute them in parallel. Therefore, instead of using database transactions, Discourse developers implement two lock namespaces for these two transactions so that locks coordinating line 6 will not interfere with locks for line 12. Note that RDBMS still executes line 6 and line 12 serially to avoid data corruption.

Optimistic ad hoc transactions can also benefit from column-based coordination—they only need to validate whether specific column values have been updated. Figure 9 shows a more

```
1 IN: o_id, ..
2 lock(order_id=o_id)
3 pays := SELECT * FROM Payments WHERE order_id=o_id
4 if pays is empty:
5     INSERT INTO Payments VALUE (o_id, ...)
6 unlock(order_id=o_id)
```

Fig. 10. Spree uses an ad hoc transaction that precisely locks the equality predicate at application level to avoid database gap locks when processing payments.

accurate representation of the edit-post transaction in Discourse [21], which we previously discussed in Section 3.1.2.[4] It performs value-based validation on the updated content column to detect concurrent changes. Any concurrent update to other columns, including view_cnt increments, will not interfere with content updates. Overall, there are five ad hoc transactions where developers use column-level coordination to unleash potential parallelism.

Gap vs. Predicate. Knowing the search conditions, developers can use the precise predicate for coordination. This can avoid false conflicts caused by the gap lock used in the major RDBMSs [68, 75, 77], including MySQL and PostgreSQL. As shown in Figure 10, in the Spree [113] e-commerce application, database systems might concurrently execute the following code with order_id of 10 and 11 corresponding to two orders created by transaction TXN 1 and TXN 2, respectively. In TXN 1, line 3 checks if any payment row exists for the order identified by order_id = 10. Since an order can have many payments (to allow mixed payment methods), the order_id index of the Payments table is non-unique. Suppose that it currently indexes values 9 and 12. Executing line 3 of TXN 1 causes the database system to acquire a gap lock on the index interval $(9, 12)$, blocking concurrent inserts to this range so that re-executing line 3 can obtain repeatable results. Meanwhile, line 5 in TXN 2 inserts a new payment row for another order whose order_id equals 11. Though this insert does not interfere with TXN 1's line 3, it would nevertheless be blocked by the gap lock. To make matters worse, this situation can be commonplace in e-commerce applications. Check-out operations are usually performed on newly created orders, which have the largest order_ids. Such operations would content on one common interval—the one starting from the latest paid order's order_id to infinity—and therefore block each other. We consider these locks a variant of *predicate locks* [36, 58], as they use predicate information of accesses (i.e., the order_id values) to achieve precise mutual exclusion without false conflicts. Among the 91 cases we studied, 10 cases implement predicate locking for accurate coordination, all based on equality predicates; 1 case implements both column-based coordination and predicate-based coordination. Predicate locking can be achieved with a concurrent hash table tracking locked values for simple equality predicates. Since developers understand web applications' accesses better than database systems, it is more practical for them to derive a customized predicate locking scheme than for database systems to provide general predicate locking.

Discussion. Both predicate locking and column-level locking introduce performance costs to database systems. For complex predicates, the performance advantage of ad hoc transactions might diminish due to the cost of deciding predicate compatibility. The cost grows with the generality of supported predicates, and expensive **satisfiability modulo theories (SMT)** solvers would be ultimately required. For example, to support range predicates, an intuitive method is to store all active ranges in an interval tree. In this case, ad hoc transaction performance would depend on the

[4] However, the version column still exists for use in other APIs.

performance and scalability of the underlying tree structures, to obtain which require significant effort [69]. For column-level locks, the main cost is space usage, as each column requires a lock.

3.4 How are Failures Handled?

Similar to database transactions, ad hoc transactions should handle runtime failures, such as deadlocks and failed validation, as well as system crashes, such as database server crashes and web server crashes.

> FINDING 5. *Ad hoc transactions typically do not have complex failure-handling logic, partly because there are fewer failure scenarios that need to be handled (e.g., the absence of deadlocks) and partly because developers seem to often assume failure-free executions.*

3.4.1 Automated Rollback vs. Manual Rollback. We first consider failures without any crashes. These failures are usually caused by deadlocks or validation failures and are traditionally handled by database rollback mechanisms. Unlike database systems that provide a general catch-all rollback mechanism, application developers need to craft failure-handling logic on a case-by-case basis, just as how they design the coordination for each ad hoc transaction.

Deadlocks. For deadlocks, we have seen no deadlock detection and handling logic. In pessimistic ad hoc transactions, we find that either a single lock is used (52/65 cases) or locks are acquired in a consistent order (13/65 cases). Thus, none of them needs to handle deadlock at runtime. The same applies to locking in optimistic ad hoc transactions. In addition, some optimistic cases do not acquire any lock during the validate-and-commit process, which obviously eliminates deadlocks but sacrifices correctness. We discuss these correctness issues in Section 4.1.2.

Validation Failures. Meanwhile, validation failures happen only in optimistic ad hoc transactions. We find that 19 cases directly return an error to end users on validation failures without persisting any update. In other cases, non-critical updates are issued before the validation phases, which requires rollbacks upon validation failures. Optimistic ad hoc transactions either use certain *rollback methods* to negate the effect of updates or use *repair techniques* to “roll forward” and commit changes, as discussed below.

Rollback methods in ad hoc transactions are either based on (i) database transactions’ atomicity property or (2) hand-crafted rollback procedures. There is one case using the former method. It uses a database transaction with Read Committed isolation to enclose update and validation statements. A user-initiated abort is issued to terminate the database transaction and roll back updates if the validation fails. Meanwhile, two cases are equipped with manually written rollback procedures. These procedures are triggered by validation failures and will undo persisted updates. Interestingly, we noticed that applications may not always undo all changes made in an ad hoc transaction when they roll back. For example, Broadleaf employs an ad hoc transaction to avoid concurrent handling of the same order, which spans almost the whole processing procedure. If the validation on the SKU state fails, although prior updates to payment and order status will be undone, other changes, such as the update to total order price, will not be rolled back. This is an example of ad hoc transactions with relaxed semantics in terms of failure handling, instead of having stringent all-or-nothing atomicity restriction like database transactions.

Repair techniques are used in four cases to handle conflicts, which fixes inconsistent values instead of rolling back the whole transaction. This idea relies on developers’ knowledge of program dependency and is similar to the transaction repair optimizations [25, 122]. Consider the example shown in Figure 11, which is taken from the Discourse [21] forum application, a periodic

```
IN: original, shrunken
posts := SELECT * FROM Posts WHERE img_id=original.id
for post in posts:
  while true:
    new := replace(original, shrunken, post.content)
    success := UPDATE Posts
                SET content=new, img_id=shrunken.id, ver=ver+1
                WHERE id=post.id, ver=post.ver
    if success: break
    post := SELECT * FROM Posts WHERE id=post.id
```

Fig. 11. Discourse uses an ad hoc transaction that repairs affected post updates upon conflicts, instead of rolling back the whole transaction.

background task that shrinks large images in posts. Since multiple posts can use the same image, this transaction may conflict with a user-initiated post edit, which only modifies a single post. In such cases, an database system may abort the transaction and rollback work done for other unaffected posts, and the application has to perform shrinking and content replacement again. A better solution is to identify the changed post, only redo the content replacement for it, and commit the image shrinking transaction.

3.4.2 Crash Handling. We mainly focus on two types of crashes: (i) database server crashes and (ii) application server crashes. We exclude client-side failures, such as force shutdown of a browser, as they have no impact on the ad hoc transactions that we studied. In the applications we examined, clients do not make direct database access. However, we note that with advancements in techniques like **progressive web application (PWA)** and business models like **mobile backend as a service (MBaaS)**, future web applications will likely offload heavy business logic (in part) to the client side and enable the client to access database systems and other storage systems directly. In that case, client-side failures will become relevant and add to the complexity of the design and implementation of ad hoc transactions.

Database Server Crashes. When the database server crashes, application server-side database drivers will detect connection loss and throw runtime exceptions to notify the application to perform failure handling after database system recovery. To gracefully handle such failures, applications need to wait until database connections re-establish and then either proceed with the interrupted business logic or roll back as we previously discussed. However, we find that no ad hoc transaction performs error handling in this way; they simply let the exception propagate to the web framework and ultimately display an internal error page to end users. Since no rollback is performed, the database might be left in an intermediate state. We find that in some cases ad hoc transactions can tolerate such intermediate states with preventative measures. For example, the check-out procedure in Broadleaf creates a payment record for the order and a crash during check-out can leave this record in an unconfirmed state. To avoid creating duplicate payment records when users attempt to check out the order again, Broadleaf issues an Update statement to set all existing unconfirmed payment records associated with the order to an archived state. This ensures that further processing treats only one payment record as active and does not double charge this order. Note that the premise of such preventative measures is that developers have to anticipate and plan for potential crash scenarios, which can be a difficult and incomplete process in practice.

Application Server Crashes. However, when the application server crashes, rollback statements for ongoing ad hoc transactions cannot be issued. To correctly resume service after application reboot, applications must ensure that changes to both the coordination metadata and the rest of the application data are restored properly.

Restoring coordination metadata is relatively easy. Versions stored in separate database columns and used in optimistic ad hoc transactions can be handled trivially. If some version is incremented by an unfinished ad hoc transaction, it can simply retain the new incremented value. The restarted application server can always reread the latest version and reinitiate the optimistic ad hoc transaction when detecting mismatching versions.[5] Locks, on the other hand, can be tricky. They might remain in a locked state while the owner thread dies when the application server crashes, which can cause deadlocks later. Fortunately, most lock metadata used in ad hoc transactions does not persist forever—they either vanish along with crashes (in-memory locks) or expire after a given period (Redis locks). There is one exceptional case in Broadleaf [18], which uses locks persisted in a database table as shown in Figure 7(a). To avoid deadlock, developers associate each lock with a boot-time generated **universally unique identifier (UUID)**, which is shown as the run_id variable, that distinguishes each run. Thus, Broadleaf can ignore prior unreleased locks after reboot by examining the saved UUIDs.

Restoring other application data can be challenging as it requires tracking and persisting API progress during the rollback or forward execution. However, we have observed that no application has made this effort and it is thus difficult for the restarted application to determine the updates have persisted before the crash. This differs from rolling back on validation failures discussed in Section 3.4.1, where developers decide on the rollback points. Unsurprisingly, we have found that no application rolls back changes made by an unfinished ad hoc transaction after it crashes and restarts, which confirms our observation. Therefore, to deal with intermediate states, applications need to take preventative measure similar to how they handle database server crashes.

Interestingly, we found that developers have written database consistency checkers, similar to fsck for file systems, periodically invoked when the application is online. For example, Discourse [21] checks and fixes inconsistent references, such as missing avatars, thumbnails, and topics every 12 hours. However, whether these checks are sufficient to ensure (eventual) recoveries to a consistent state is in question. Combined with the fact that many cases skip rollback (Section 3.4.1), it can be indicated that some applications are designed to tolerate intermediate states to a certain extent. We discuss issues caused by intermediate states in Section 4.

3.5 Comparison with Database Transactions

Developers use ad hoc transactions to achieve the same ultimate goal as database transactions: correctly execute application logic in the face of concurrency and errors [45]. Furthermore, both approaches achieve this by coordinating concurrent units of work comprising multiple data operations. To gain a deeper understanding of the similarities and differences between these approaches we further compare in this section their conceptual details, e.g., what defines transaction scopes, and their semantic details, e.g., what is guaranteed. Table 5 gives a summary.

Concepts. Although there is not a well specified language like SQL for describing ad hoc transactions, many concepts in database transactions find their analogies in ad hoc transactions, as shown in Table 5, which allows a detailed dissection of ad hoc transactions. A database transaction is a sequence of database operations that transform database states in a consistent manner. It is defined by the Transaction Start, Commit, and Abort statements as well as query and data modification statements enclosed in them. In contrast, an ad hoc transaction is a piece of application code that access shared states, including but not limited to database states, to perform

[5] Even if the old value is remembered at the client side (e.g., for a multi-request editing process).

Table 5. Comparison between Database Transactions and Ad Hoc Transactions

Aspect	**Database transactions**	**Ad hoc transactions**
Forms	A sequence of database operations given in a single database connection.	A piece of application code that accesses shared states, including database states.
Scopes	Explicitly decided by begin, commit, and abort statements.	Implicitly decided by developers, though mostly inferable from application context such as lock usage.
Coordinated operations	All database operations inside the scope.	Specific operations, including accesses to non-database states.
Coordination methods	With general concurrency control and logging protocols in the database system.	With synchronization primitives manually installed by application developers.
What it means to commit	An explicit issuance of a commit statement and its successful execution at the database system.	Certain executions of the application code following execution paths that developers (implicitly) consider successful.
Atomicity	All or nothing: either all effects are visible or none at all.	Developers decide at will whether effects should be undone and how upon failures.
Consistency	Both require developers to ensure transactions transform application states correctly.	
Isolation	Standardized isolation levels, such as Serializable and Read Committed.	Developers decide at will how isolation should be achieved, sometimes in non-standard way.
Durability	Committed effects survive failures.	Developers decide at will whether data is persisted before commit.

business logic in a consistent manner. To execute database transactions, database systems employ system-level concurrency control and logging protocols, which are transparent to applications, to ensure the ACID properties. Due to the ACID execution, if an application correctly constructs its business logic into database transactions, its correctness can be easily ensured. However, for ad hoc transactions, application developers need to manually install synchronization primitives among application code without (fully) resorting to database transaction mechanisms. Without a system-level coordination mechanism, developers need to design and implement the coordination on a per-case basis. Consequently, only the operations that developers specifically attended to will be coordinated. Therefore, ad hoc transactions permits much more coordination flexibility at the potentially higher cost of development and maintenance.

The most distinctive trait of ad hoc transactions is the role played by developers' intentions in defining transaction behaviors: without a written contract, we as outside observers can only speculate what developers intend ad hoc transactions to be. For example, although the concept of transaction scopes applies to ad hoc transactions, it is however implicitly defined by developers, and we can only infer their scope according to the application source code: locks' critical sections and spans of validation procedure starting from the corresponding first reads. Since these primitives might be wrongly placed and cause errors, therefore we cannot treat them as oracles for the scopes of ad hoc transactions. Furthermore, outcomes of ad hoc transactions are either succeeded or failed, which is the same as database transactions, but they depend on how developers interpret ad hoc transactions' concrete executions, since there is no longer explicit Commit/Abort statements. When an ad hoc transaction ends, e.g., finishing its final unlock() call, outside observers can only guess, based on the visible effects, whether or not developers would consider this execution successful. Such a lack of clarity might add to the overhead of developing and maintaining ad hoc transactions and lead to subtle errors, such as omitting critical operations from ad hoc coordination (Section 4).

Semantics. We next compare the semantic differences between the two approaches through the lens of ACID [51].

The most interesting differences lie in the I of ACID: *isolation*. The classic isolation property dictates that actions of one database transaction will run in isolation with other database transactions [36]. Based on this definition, one major difference that we have already seen is that only part of the operations in ad hoc transactions will be coordinated and thus isolated from other ad hoc transactions. Another difference is that the coordinated operations in one ad hoc transaction are not isolated from *all* other ad hoc transactions, but rather a subset of ad hoc transactions that the developers are concerned with. These are natural consequences of developers manually installing synchronization primitives to coordinate ad hoc transactions. While these differences might cause ad hoc transactions to be more error-prone, as developers can easily omit APIs that perform conflicting data accesses (Section 4.2), they also suggest potential research directions for optimizing database-backed applications. For example, one might devise coordination schemes that independent transaction coordinators are spawned for each disjoint set of ad hoc transactions to avoid potential hot spots. We discuss hints on future research in more details in Section 6.

Furthermore, we find that developers have used ad hoc transactions to achieve isolation different from all four standardized isolation levels. This is achieved by using value-based validation in optimistic ad hoc transactions. For example, in Figure 9, the Discourse forum application validates whether updates can be applied to a post by comparing the current value of the post's content field with the one retrieved from a previous request (Section 3.1.2). This ad hoc transaction allows the following non-Serializable history, using the notations of [1].

$$H : w_0(x_0), c_0, r_1(x_0), r_2(x_0), r_2(x_0), w_2(x_2), c_2, r_3(x_2), r_3(x_2), w_3(x_3), c_3, r_1(x_3), w_1(x_1), c_1.$$

In this history, transaction T_0 creates the post version x_0 (the subscript refers to the transaction that writes the version) and transactions T_1, T_2, and T_3 are three post editing processes initiated by individual users, say, u_1, u_2, and u_3. In the corresponding execution, u_1 is the first to start the post edit, fetching the post version x_0. However, while u_0 is editing locally after his first request, $r_1(x_0)$, u_2 and u_3 started and submitted their edits sequentially, leaving the post at version x_3. Although x_0 and x_3 are different versions from the execution perspective, in this particular history, their values are the same. Since the ad hoc transaction uses value-based validation (Figure 9), u_1 can still successfully submit his edit, leaving the post at version x_1. Interestingly, H is not Serializable because there are always cycles (with anti-dependencies) in H's serialization graph given any arbitrary ordering of the four post versions.[6] This ad hoc transaction's isolation is also not the same as other weak isolation levels, such as Read Committed, because histories permitted in those levels are not all permitted by developers' manual coordination. Such non-Serializable executions do not appear to hinder application correctness: it is unlikely to cause issues if a user is unaware that the content has been changed and then reset while he is editing. Such a scenario is an instance of the infamous ABA problem that general purpose CC needs to prevent. However, as developers are equipped with application-specific domain knowledge, they can explicitly choose to not handle such benign ABA problems.

Next, we discuss the other three properties of ACID. We begin with *atomicity*, which means that the effects of a transaction occur either entirely or not at all, even if failures interrupted its execution. As shown in Section 3.4, ad hoc transactions we studied strive to ensure atomicity

[6] Specifically, H is not *conflict* serializable under the formalism of [1, 2]. Also, it is not *view* serializable using the formalism of [17].

Table 6. Categorization of Incorrect Ad Hoc Transactions

Category	Description	Apps	Cases
Incorrect synchronization primitives	Locking primitive impl./usage issues.	6	36
	Non-atomic validate-and-commit.	3	11
Incorrect ad hoc transaction scope	Omitting critical operations.	4	11
	Forgetting ad hoc transactions.	3	5
Incorrect failure handling	Incomplete transaction repair.	1	1
	Not rolling back after crashes.	1	3

Note that one ad hoc transaction can have multiple issues.

when runtime failures occur. However, most have not catered to either server crashes or database system crashes. As a result, when such failures occur, the effects of ad hoc transactions are left in an intermediate state, violating the atomicity requirement. As will be shown in Section 4.3, not providing crash-atomicity can hinder both application correctness and user experience.

Durability means the committed effects of a transaction are sure to remain. Failing to provide durability is similar to failing to provide atomicity, as both leave applications in an intermediate state. In ad hoc transactions, durability heavily relies on how the application interacts with different storage systems. When persisting data only into a database system, which typically executes individual database operations as single-statement auto-commit transactions, the durability of the ad hoc transaction is trivially ensured: an ad hoc transaction commits after its writes have returned, which are made durable by the database system. However, when using other storage systems, durability can be tricky. For example, Redis, by default, only periodically persists snapshots to disks, and a crash may cause loss of changes made in recent minutes. To achieve durability, developers need to enable logging explicitly, and we have found no evidence that developers have done so in the studied applications. Similarly, making local file system updates durable can also be tricky. Instead of updating files in place, which requires explicit flushes for durability, we find that developers usually resort to the “safe rename” pattern: they append the file content to a new file and rename it as the target file, which in many (unfortunately not all) file systems persists all writes and ensures durability once renaming succeeds [84]. Thus, no durability issue is found in dealing with local file systems.

Finally, *consistency* is more of a requirement for developers than a property: transactions should have no logical errors and correctly transform the application state from one consistent state to another. Although some applications run background consistency checkers, which might tolerate incorrect state transformation of ad hoc transactions in certain cases, we nevertheless consider the consistency requirement is imposed on developers regardless of which coordination approach they choose.

4 CORRECTNESS ISSUES

The variety of implementation possibilities as we discuss in Section 3 indicates that building correct ad hoc transactions is nontrivial. This section examines the correctness issues of ad hoc transactions and relates them to the design characteristics. Due to the manual nature of our examination, the issues discussed below might be incomplete. Systematic methods for detecting ad hoc transaction issues are left for future work.

Result Summary. 69 correctness issues are found in 53 cases (Table 6); some cases have multiple issues. We have manually verified that all issues are reproducible and cause user-noticeable consequences. Among them, 28 cases have severe consequences (Table 7), such as charging cus-

Table 7. Incorrect Ad Hoc Transactions can have Severe Consequences

Application	Known severe consequences	Cases
Discourse	Overwritten post contents, page rendering failure, excessive notifications.	6
Mastodon	Showing deleted posts, corrupted account info., incorrect polls.	4
Spree	Overcharging, inconsistent stock level, inconsistent order status, selling discontinued products.	9
Broadleaf	Promotion overuse, inconsistent stock level, inconsistent order status, overselling.	6
Saleor	Overcharging.	3

tomers incorrect amounts, while others cause less fatal results, such as displaying stale content. Most issues relate to the primitives' usage and implementations (49/69), while others occur in the choosing of what to coordinate (16/69) and handling abort (4/69). We have submitted 20 issue reports (covering 46 cases[7]) to developer communities; 7 of them (covering 33 cases) have been acknowledged.

4.1 Incorrect Locks and Validation Procedures

FINDING 6. *36 out of 65 pessimistic ad hoc transactions incorrectly implement or use locking primitives; 11 out of 26 optimistic ad hoc transactions do not ensure the atomicity of validation and commit, causing correctness issues.*

4.1.1 Locking Primitive Issues. There are seven different lock implementations (Section 3.2.1) and five of them can be incorrect.

Incorrect Lock Usage. When developers reuse existing systems' locking primitives, misuses arise. There are two existing locking primitives reused, database systems' Select For Update statements and Java's synchronized keyword (Section 3.2.1), and both have corresponding cases of incorrect usage. Spree [113] serves as an example of incorrectly using the Select For Update statement. Since the lock acquired by Select For Update statements is released when the current transaction commits, developers need to ensure that critical operations are executed within the current transaction. Unfortunately, Spree does not explicitly enclose the Select For Update inside a database transaction, which causes the database lock to release as soon as the statement returns [89]. Meanwhile, SCM Suite [33] shows an interesting issue related to the synchronized keyword. After loading data from the database system, SCM Suite uses this keyword to synchronize over thread-local ORM-mapped objects. As a result, conflicting threads acquire different locks and can never block each other [130].

Another type of misuse happens when developers intend to use a single lock to coordinate RMW operations: they omit the coordination on the first query statement. Specifically, though ad hoc transactions intend to acquire locks to coordinate all RMW data accesses, sometimes the lock key, e.g., an ID, is known after the data is fetched. In these situations, developers need to re-read the data after acquiring the lock to coordinate the entire RMW. There are 2 cases where the developers forget the re-read, leaving the initial read in RMW uncoordinated. As shown in Figure 12, an ad hoc transaction in Spree uses locks to coordinate concurrent payment processing. However, the

[7] Some affected cases can be resolved in one code patch.

```
IN: order_id
payments := SELECT * FROM Payments WHERE o_id=order_id
for payment in payments:
    lock(payment.id)
    if payment.state != 'completed':
        process(payment) // Deducts from store credits or other sources.
        UPDATE Payments SET state='completed' WHERE o_id=order_id
    unlock(payment.id)
```

Fig. 12. An incorrect ad hoc transaction in Spree for payment processing whose reading of payment status is excluded from the payment lock scope.

```
IN: id, version
ActiveRecord.transaction do
    result := MiniSql.query(
        "UPDATE Reviewables SET version=version+1
        WHERE id=id AND version=version RETURNING version")
    if result is null: raise UpdateConflict exception
    /* Perform actual operation on reviewables. */
end
```

Fig. 13. An incorrect optimistic ad hoc transaction from Discourse that fails to ensure the atomicity of validation and commit, due to the incompatibility between ActiveRecord and MiniSql.

payment is locked only after being read from the database system during payment processing, as only the order ID is given. Although the lock serializes the subsequent writes, the entire RMW process is not atomic, which may result in the processing of a payment twice and overcharging a user if duplicate checkout requests are sent [92].

Incorrect Lock Implementations. The locking primitives implemented by developers can also have correctness issues. Specifically, developers incorrectly build the locking primitives with Redis store and in-memory lock tables (Section 3.2.1). For lock based on Redis, Mastodon [106] gives an example where the developers implement the lease semantics. Specifically, they enable the auto-expire feature of Redis [94] for lock entries. As a result, the lock might be released early when the entry times out before the coordinated critical section finishes. Unfortunately, Mastodon does not check whether the lock has expired early and experiences inconsistency, such as deleted posts appearing in followers' timelines [23]. Furthermore, all ad hoc transactions in Mastodon are based on this incorrect lock implementation. For lock based on in-memory lock table, Broadleaf [18]'s eviction-enabled lock table also provides lease semantics—when table size reaches a given limit, an LRU policy is invoked to evict locks from the table [95]. As a result, if a lock held by the transaction is evicted, two conflicting transactions (e.g., check-out and add-cart) may concurrently access the same data (e.g., the order total), which causes inconsistency such as users not paying for concurrently added items.

4.1.2 Non-Atomic Validate-and-Commit. Validation-based optimistic ad hoc transactions need to avoid conflicting updates between validation and commit. Thus, they need to guarantee validate-and-commit atomicity. However, atomicity violation happens when developers manually implement validation procedures (16 cases), while ad hoc transactions using ORM-generated validation procedures ensure atomicity (10 cases). Figure 13 shows such an example from Discourse [21]. In this example, versions are used to track changes to reviewable items (e.g.,

```
IN: order_id
lock(order_id)
skus := /* Select relevant SKUs for the given order via order_id. */
/* Process the order, including payment, shipping, etc. */
for sku in skus:
    Update SKUs Set ... Where id=sku.id
unlock(order_id)
```

Fig. 14. An incorrect ad hoc transaction in Broadleaf's check-out API that misses the coordination for SKU-related operations.

a controversial topic) and prevent conflicting administrator operations. Developers explicitly enclose the validation (line 4–6) and subsequent updates (line 7) in an Active Record transaction block, within which queries *should* be issued in a database transaction. However, the validation queries are expressed using interfaces provided by MiniSql [22], a module independent of Active Record. As a result, Active Record cannot intercept and issue validation queries as part of the database transaction, thus failing to provide validate-and-commit atomicity [90].

4.2 Incorrect Coordination Scopes

Incorrect coordination scopes refer to errors developers make when choosing what to coordinate in ad hoc transactions.

> Finding 7. *16 issues arise from incorrect coordination scope. Specifically, developers either omit some critical operations in existing ad hoc transactions (11/16) or forget to employ ad hoc transactions for certain business procedures altogether (5/16).*

Omitting Critical Operations. Though the flexibility of choosing what to coordinate is an advantage of ad hoc transactions (Section 3.1.1), it comes with an increased chance of leaving critical operations uncoordinated. As shown in Figure 14, the ad hoc transaction in Broadleaf [18] that coordinates the check-out process omits coordination for all SKU-related operations. As a result, concurrent check-outs of different orders that purchase the same SKU can lead to inconsistency between the SKU quantity decrement and the number of sold items [96]. Furthermore, this bug is less apparent than it is depicted in the simplified example. The locking over order IDs is achieved via a Spring request filter that examines all incoming requests and acquires an order lock if an order_id argument is present in the request parameters. Meanwhile, the SKU modification is buried in a specific request handler. It can be difficult for a request handler developer to understand how other application-level synchronization constructs may (or may not) perform related coordination. Optimistic ad hoc transactions are subject to such errors as well. For example, in Spree [113], the transaction that decrements SKU quantity (shown in Section 3.1.1) also involves setting the order status column. However, modification to order status is not coordinated, allowing duplicate decrements and resulting in inconsistent stock levels [89].

Forgetting Ad Hoc Transactions. Forgetting to coordinate certain business logic with transactions is a general problem with both ad hoc and database transactions. However, it is more disastrous with ad hoc transactions. A conflicting business procedure without proper ad hoc transactions installed (e.g., another request handler) can freely interleave with other procedures coordinated by ad hoc transactions, reading and writing "coordinated" data. For example, in Spree [113], all ad hoc transactions are deployed in the request handlers that return responses in the HTML format. However, another uncoordinated set of handlers with the same functionality exists and produces JSON format responses. As a result, JSON handlers' interleaving with HTML handlers leaves database system states inconsistent [87]. To detect such issues, developers have to understand how concurrent threads of handler execution conflict with each other and know all conflicting operations of a specific handler.

4.3 Incorrect Failure Handling

> FINDING 8. *Ad hoc transactions might incorrectly handle failures, including both runtime failures and server crashes.*

Incomplete Repairs. When using transaction repair to "roll forward" an affected transaction, developers might derive an incomplete repair, such that not all affected operations are re-executed. In Discourse [21], when updating image references of posts, developers use versions to track individual states of fetched posts from a query (pseudocode shown in Section 3.4.1). Though concurrent modification to a specific post can be precisely detected and repaired, newly added posts that qualify the query are neglected. As a result, those new posts will not be processed, and their image references are thus dangling, presented as broken links to end-users [93]. This is the only case that has this issue.

Unexpected Intermediate States after Crashes. If an application is not designed to tolerate intermediate database states and rollback handlers fail to prevent such states, the application might fail to provide normal services if server or database crashes occur. We thoroughly investigated the impact of crashes in ad hoc transactions from Broadleaf and Spree. We identified 31 unique crash scenarios where a crash leaves writes partially executed and found that crashing at 28 among those (90.3%) have to user-noticeable consequences. If crashes injected at different code locations lead to the same set of partially executed writes, we consider them as identical crash scenarios.

We briefly describe one such bug. In Spree [113], a server crash during check-out can leave payments in an intermediate state (i.e., having the status column equalling 'processing'). Since such payment status values are not rolled back after reboot, Spree can neither initiate new payment operations due to the unfinished ones nor resume payments initiated before the crash because they are considered to be "processing" by active threads. Therefore, users can never finish the check-out [88].

Note that database crashes also result in the same errors when injected at code locations where a server crash leads to errors, since neither application handles database disconnection exceptions. However, the inverse is not true: database crashes can introduce additional errors that are not caused by server crashes. For example, when using Broadleaf's database-backed order locks (Figure 7(a)), a database crash during the lock's critical section leaves the lock in an acquired state. Although Broadleaf uses boot-time generated UUIDs to distinguish between unreleased locks from previous runs and those currently acquired, a database crash does not trigger regeneration of the current UUID. As a result, Broadleaf cannot acquire the order look during whose critical section the database system has crashed until the application reboots.

For those benign[8] crash scenarios (i.e., those without user-noticeable consequences), we have identified two reasons. In two benign crash scenarios in Broadleaf, although the application state is left inconsistent, Broadleaf handles such inconsistencies with a preventative measure that marks

[8] We used a loose criterion for "benign" crash scenarios. We consider a crash scenario benign as long as applications could continue to serve normally after reboot, even if user requests active at crash time cannot receive a proper response (which to users are indistinguishable with other errors such as bad network connection).

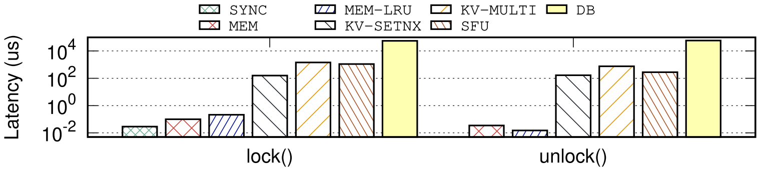


Fig. 15. Latencies of different lock implementations. The invocation latency of synchronized is shown under lock().

those partial writes as "archived," as described in Section 3.4.2. Thus, it could continue to serve normally after reboot. In another benign crash scenario from an optimistic ad hoc transaction in Spree, the writes before the crash only affect the coordination metadata, namely the version counter. Therefore, Spree could normally serve by re-executing the ad hoc transaction based on the new version.

5 PERFORMANCE EVALUATION

This section further investigates the performance of different designs and implementations of ad hoc transactions using actual application codebases.

Result Summary. First, there are order-of-magnitude performance differences between different primitive implementations. Disk I/Os and network round trips are the decisive factors. Second, all four customized coordination granularities benefit API performance. Ad hoc transactions perform up to 1.3× better than database transactions in contentious workloads and similarly in no contention workloads. Third, for rollback performance, transaction repair achieves the lowest latency among other rollback methods.

Experiment Setup. For API performance, we developed test clients to stress chosen application APIs with valid HTTP requests; for primitive performance, we reused applications' original implementations. Applications are tuned according to official guides and deployed separately from the test client. We use either MySQL 8.0.25/5.7.36 or PostgreSQL 13.5, whichever is defaulted or recommenced, as the backing RDBMSs, separately deployed and carefully tuned. Each machine has 2 x 12 2.20 GHz physical cores (Intel Xeon Processor E5-2650 v4), 128 GiB DDR4 memory, and a 1 Gbit/s NIC.

5.1 Different Primitive Implementations

We ported all lock implementations to either Java or Ruby microbenchmarks and evaluated their latencies with a simple workload where a client repeatedly invokes lock() and unlock() in a loop.[9] Figure 15 shows the results. The latency differences are of orders of magnitude. The slowest among them is the RDBMS-based one (DB), ported from Broadleaf [18], where it performs within a database transaction first a Select to check if the existence of a corresponding lock row and then an Update or Insert to acquire the lock. Since the RDBMS needs to flush writes for durability, and this lock has the highest latency. The Redis-based locks (KV-SETNX, KV-MULTI), ported from Mastodon [106], Discourse [21], and Saleor [108], and the Select For Update–based locks (SFU) all have millisecond-level latencies. They are much faster than DB because their locking logic does not involve expensive disk I/O. Interestingly, KV-SETNX is also faster than KV-MULTI because the former only issues a single Redis command, while the latter sequentially issues seven (Section 3.2.1). Finally, by eliminating all network round trips, those in-memory locks, i.e., map-based locks (MEM and MEM-LRU), ported from Broadleaf, and the synchronized keyword (SYNC) have the best performance.

[9] We skip the evaluation for validation-based implementations because they mainly differ in the locks that ensure atomicity.

Table 8. APIs and Setups for Evaluating Coordination Granularities

Granularity	**Application API(s)**	**Workload** (with contention)	**RDBMS**	DBT **isolation**
RMW (Section 3.3.1)	check-out, Broadleaf [18]	Customers purchase the same SKU.	MySQL	Serializable
AA (Section 3.3.1)	like-post, Discourse [21]	Users like different posts of seven contented topics.	PostgreSQL	Serializable
CBC (Section 3.3.2)	create-post & toggle-answer, Discourse [21]	Assign distinct topics to user pairs, where one user creates posts and one accepts answer.	PostgreSQL	Repeatable Read
PBC (Section 3.3.2)	add-payment, Spree [113]	Customers submit payment options for new orders.	PostgreSQL	Serializable

We obtain no-contention workloads by switching users to work with different SKUs/topics or existing orders.

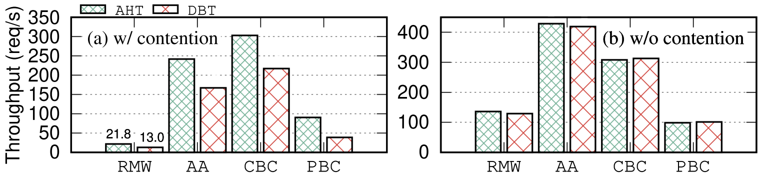


Fig. 16. API throughputs using different coordination granularities.

5.2 Different Coordination Granularities

Ad hoc transactions can perform coordination at granularities rarely seen in database systems (Section 3.3). To understand their impact, we chose and evaluated four real-world APIs, where the four granularities discussed earlier are employed, denoted as RMW (read–modify–write), AA (associated access), CBC (column-based coordination), and PBC (predicate-based coordination).

We first measure each API's throughput with the original, ad hoc transaction–based codebase (denoted as AHT) and a modified one using database transactions with the weakest yet sufficient[10] isolation level instead (denoted as DBT). Table 8 lists the specific APIs, workloads, and setups. APIs used in CBC and PBC are previously described in Section 3.3.2; RMW's API is similar to the one described in Section 3.1.1 but excludes unnecessary timestamp updates; AA's like-post API increments the given post's like count and updates its parent topic's total like count. Figure 16 shows the peak throughput of each API. Under contentious workloads, AHT achieves up to 1.3× higher throughput than DBT and the geometric mean of improvements is 63.0%. Under no-contention workloads, AHT and DBT have similar performance. These results confirm our hypothesis on the potential benefits of using customized coordination granularities. Specifically, in RMW and AA, acquiring locks early and aggressively prevents deadlocks in MySQL and write–write conflicts in PostgreSQL. As a result, conflicting API requests' non-critical sections are effectively pipelined with the one active critical section, improving CPU efficiency. Meanwhile, by coordinating at a more fine-grained and precise level, CBC and PBC avoid false conflicts of database transactions. Therefore, more transactions can be processed and committed in AHT than in DBT. In terms of scalability, both codebases exhibit similar throughput plateaus for each API, except for PBC, albeit DBT's throughput increases at a slower rate. In PBC, the DBT throughput reaches a plateau at around

[10] By sufficiency, we mean an isolation level prevents application inconsistency caused by anomalies such as lost updates or phantom reads.

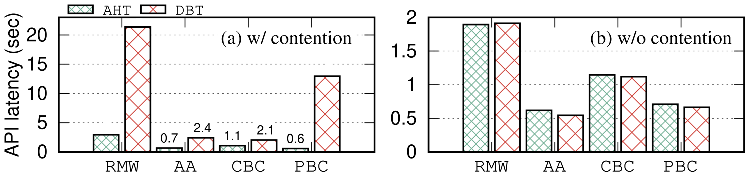


Fig. 17. The 99th percentile API latency using different coordination granularities with 48 client threads.

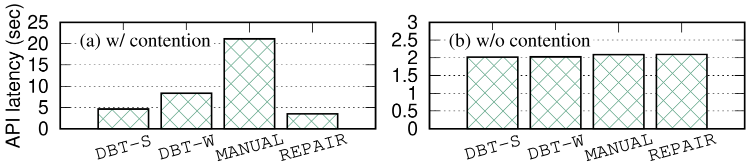


Fig. 18. API latencies using different rollback methods.

8 clients, while the throughput with AHT continues to scale up to 44 clients. This is because, in this workload, the predicate lock in PostgreSQL (i.e., page locks) effectively causes each database transaction to conflict with each other. Therefore, the increase in throughput is solely due to overlapping network communication and transaction execution, and the database system can process and commit only one transaction at a time. We have omitted the figures for scalability results for brevity. Under non-contentious workloads, both codebases have similar performance in four APIs.

We then evaluate the latency of both codebases. We measure the complete interval in which a client successfully performs an action (e.g., placing an order), including retries in case of API call failures due to conflicts. When there is only a single client thread (or two in CBC, each corresponding to one API), DBT and AHT exhibit similar latency. Specifically, the geometric mean of the DBT-to-AHT latency ratio is 1.03 (resp., 1.05) for the median (resp., 99th percentile) with a relative standard deviation of 0.07 (resp., 0.04). However, in the case of multiple client threads, AHT shows lower tail latency than DBT in contentious workloads, as shown in Figure 17. This is because that transactions in AHT never abort once they are granted with locks, a consequence of the absence of deadlocks (Section 3.4.1), while transactions in DBT may abort due to conflicts, which necessitate API retries. For example, in RMW, the p99 lock waiting time for AHT is 2.4 s, while the p99 transaction retry count for DBT is 14 and each retry takes 1.25 s on average. For median latency and latency in non-contentious multi-client workloads, both codebases perform similarly.

5.3 Different Rollback Methods

Finally, we evaluate the performance of different rollback methods with Discourse's shrink-image API. The API and rollback methods are previously described in Section 3.4.1. The chosen API implements transaction repair to handle errors (denoted as REPAIR). We further adapt its codebase to implement rollback with Read Committed database transactions (denoted as DBT-W) and manual rollback (denoted as MANUAL). We also built a pure database transaction baseline by replacing ad hoc transactions with Serializable database transactions (denoted as DBT-S). We use a workload where one thread invokes shrink-image for different images, each used by eight posts, and on average two threads concurrently and continuously request the edit-post API (described in Section 3.1.2) over posts of each image, conflicting with shrink-image invocations.

Figure 18 shows the shrink-image latencies, with and without conflicting edit-post requests. When there are no conflicts, shrink-image has similar latencies over four configurations since time is mostly spent on image processing. However, when there are conflicts, REPAIR shows the lowest

API latency, as transaction repair can preserve the work done for unaffected posts. Surprisingly, DBT-S beats DBT-W and MANUAL with the second-lowest latency. The reason is that, in the latter two configurations, before shrink-image aborts, it is blocked for the duration of the conflicting edit-post, as the post lock used by edit-post is also used in DBT-W and MANUAL to guard the version check. MANUAL takes longer than DBT-W, as it needs to issue multiple database operations to roll back database states while DBT-W only issues one—Transaction Abort.

6 DISCUSSION

So far, we have shown that ad hoc transactions are prone to errors and are difficult to identify and understand. However, they are still widely used in web applications, mostly among critical APIs. Furthermore, application-level coordination is observed in other large-scale web applications [9, 52, 53, 86]. For example, inspired by our study, a database engineer from Alibaba shared that Taobao, China’s biggest online shopping platform, has performed extensive application-level concurrency handling and optimization, resulting in their database systems serving only large-volume yet simple workloads [121]. Rather than attributing this phenomenon to developers overlooking the power of database transactions, we believe there are concrete motivations of using ad hoc transactions. To shed light on this subject, we offer a potentially *opinionated* perspective in this section based on our experiences and understanding of ad hoc transactions and web applications.

At a high level, we believe that there is a gap between the coordination requirements of web applications today and what database systems currently offer. Our study has shown that developers often resort to ad hoc transactions to achieve coordination that is difficult to attain with database transactions (Section 3.1). In some cases, database transactions can be used, but at the cost of increased development efforts or decreased performance. For example, to achieve partial coordination over specific data access in a long business procedure (Section 3.1.1), developers may need to explicitly establish multiple database connections and manually place data access to be coordinated in a database transaction via one of the connections while placing the rest in other connections. Additionally, developers may also need to craft individual SQL statements to avoid ORM-generated SQL statements being misplaced in the database transaction. In doing so, developers effectively discard the development assistance that web frameworks strive to provide, such as automated transaction management [114, Section 1], and end up with application logic cluttered with repetitive and error-prone database transaction management code. In other cases, database transactions are insufficient. For example, a database system alone can hardly coordinate distributed transactions, which is necessary when applications use other storage backends like KV stores and file systems to manage data (Section 3.1.3) and when applications themselves are constructed and deployed in a distributed fashion. Although many database systems have supported protocols like XA [67] (though sometimes incorrectly [50]), support in other storage systems is rather limited. As a result, developers have to craft coordination manually.

The next question that naturally arises is: what has led to this gap? We believe that the answer lies in the rapid increase in complexity of web applications while transaction support from both database systems and other parties has not kept pace. With the advent of the Internet and mobile computing, web applications’ functionalities have evolved from simple information displaying to now covering almost every aspect of our daily lives, such as shopping, socializing, productivity, and entertainment. As a result, their business logic complexity has also increased correspondingly, leading to bloated transactions if they were coordinated by database transactions, which is often considered harmful to application performance [70, 96, 103, 105]. Furthermore, the sheer complexity has also forced applications to modularize and distribute. Many application functionalities have been modularized and externally serviced, such as the Elasticsearch search engine [34] and the Stripe payment service [115]. Additionally, web applications have evolved from standalone monolithic servers to disaggregated architectures such as microservices and serverless. Meanwhile, database transactions have remained relatively stable, with their interfaces and semantics that matured in the 90s still defining how applications could program database transactions today. As a result, database transactions are becoming inadequate for coordinating increasingly complex business logic in web applications, leading to the gap we face today. Although the transaction concept remains a simple and attractive tool for building complex and reliable applications, transaction support from database systems alone is far from sufficient, forcing developers to write their own ad hoc transactions.

These observations suggest that we should take a broader perspective and consider transaction support from a global point of view instead of addressing applications' problems solely within the database system. To this end, we have identified several questions that we believe are worth further investigation and may lead to new avenues of research, which are listed below.

How do transactions impact the performance of complex web applications? Most arguments against transactions we have seen are subjective and usually lack concrete numbers and clear contexts. It is critical to have a clear understanding of how performance is affected is critical to identify the right solution for real-world applications. Performance issues with transactions are typically attributed to increased transaction complexity and two-phase commit (2PC) costs in distributed transactions. However, there has yet to be a systematic investigation into the impact of these factors. It is unclear if they always introduce performance penalties, how much degradation they bring, and what causes the cost. Is it due to the extended period of time for resource locking or the increased working set? Are the causes rooted in inefficient coordination of framework implementation or inherent in ensuring the ACID semantics? For example, if the problem is usually caused by concurrency control protocols making suboptimal locking decisions at unfortunate times or granularities (Section 5.2), a possible solution could be to derive new interfaces for providing proper locking hints while preserving the ACID semantics instead of developing new transaction semantics. Based on our experience, thoroughly evaluating and understanding application performance can be non-trivial. Therefore, a systematic understanding will be of great value for future research.

Can transaction complexity be reduced? If the complexity of web applications is causing transactions to become too complex for database systems to handle, then the next practical step is to research methods to reduce the complexity of transactions in web applications. Nowadays, developers typically program data access using ORMs within application code for portability and development agility. They also rely heavily on libraries to dynamically compose transactions, such as the @Transactional annotation provided by the Spring Framework [114], which makes data access of a marked method and its callees execute in a transaction. However, without careful attention, such library-created transactions can easily become bloated with accidentally induced data access (Section 3.1.1) and difficult to rework if the application has already been released into production. In our observation, transaction complexity is often accidentally increased. For example, developers who add a @Transactional annotation may be unaware of the exact data access made in the callees, potentially caused by individual developers being unaware of the details of functions and modules developed by others. Therefore, a solution that decouples transactions from complex application structures might be attractive, as the transaction complexity could be explicitly observed and managed. Alternatively, new interfaces that discourage developers from wrapping all data access into a single transaction while still facilitating composing correct application logic could also be beneficial. In the latter case, with new interfaces, corresponding support from the underlying database systems should also be considered.

How about alternative transaction semantics? Besides reducing transaction complexity, non-ACID transaction semantics offer another opportunity to address the performance problem. As

Table 9. Coordination Hints Supported by the Top Ten Ranking RDBMSs [26]

Coordination hints	Oracle	MySQL, MariaDB	SQL Server, Azure SQL	PostgreSQL	IBM Db2
Explicit table locks	✓ They have different restrictions (e.g., syntax) and behaviors (e.g., lock modes and conflict handling).				
Explicit row locks					
Explicit user locks	✓	✓	✗	✓	✗
Other lock hints	✗	Instance lock	Priority in deadlock handling	✗	Set default granularity
Per-op isolation	✗	✗	✓	✗	✓
Savepoints	✓ They differ in syntax and duplicate name handling.				
Other trans. hints	Autonomous trans.	✗	Nested trans.	✗	✗

We skipped SQLite (ranked six) due to space constraints; it supports snapshot-based read-only transactions but none of the listed ones. We also skipped MS Access (ranked seven) as it is mainly used for office applications, supporting up to 2 GB databases and 255 concurrent users, and Apache Hive (ranked ten) as it does not support transactions.

our study and others [52, 53, 86, 121] have observed, many transactions in real-world applications have already departed from the classic ACID semantics. For example, many ad hoc transactions have sacrificed crash-atomicity (Section 3.5). In web applications with disaggregated architectures such as microservice, the **tentative/cancel/confirm (TCC)** pattern, an interesting pattern for writing transactions across different components, has become a common practice to replace distributed ACID transactions [52, 53]. However, the state of the practice is far from ideal. Ad hoc transactions are mostly crafted from scratch and are thus error-prone (Section 4). Meanwhile, although the TCC pattern has some library support [6, 35], developers are still imposed with burdens such as the responsibility of ensuring the idempotency of individual activities' invocation and compensation [109]. As our study shows, shifting the coordination towards the application level generally forces developers to deal with problems that are only mildly related to business logic and results in applications that are prone to errors. Therefore, much work is urgently needed to formalize these non-ACID semantics, evaluate their merits and pitfalls, and develop support interfaces and systems that alleviate programming errors. Additionally, since non-ACID coordination usually happens at the application level and is not observed by database systems, research on holistic optimization from the database system to the application level is also promising.

Coordination Hints in Existing Database Systems

Our suggestions for future research are cautiously pragmatic, following a measure-then-build mindset. Nevertheless, they have prompted us to rethink the support that database systems provide for concurrency handling in web applications. In addition to standard database transactions, many existing database systems offer vendor-specific interfaces for passing hints that customize š. For example, PostgreSQL provides explicit user locks, where locks are identified by user-specified integers and scoped by the active session or transaction [49, Section 13.3.5]. These hints provide a starting point for examining how database systems can cater to the coordination requirements of web applications today.

Can these coordination hints help developers achieve the coordination goals (e.g., assist in writing ad hoc transactions or even replace them)? We compiled a summary (Table 9) of supported

Table 10. Relationship between Coordination Hints and Ad Hoc Transitions

Coord. hints	**Can potentially support**	**Can potentially avoid**
Explicit table locks	Coarse-grained coord. (Section 3.3.1)	Incorrect lock impl. and ORM-related misuses (Section 4.1.1); incorrect failure handling (Section 4.3).
Explicit row locks	Coarse-grained coord. (Section 3.3.1) and partial coord. (Section 3.1.1)†	
Per-op isolation		
Explicit user locks	Fine-grained coord. (Section 3.3.2) and non-db op. (Section 3.1.3)	Incorrect lock impl. and transaction-related misuses (Section 4.1.1).

†Work in conjunction with database transactions.

coordination hints among the top ten ranking RDBMSs [26] and found that they can in part prevent errors while retaining benefits of ad hoc transactions (Table 10). For example, to coordinate only specific database operations (Section 3.1.1), we can augment them with the HOLDLOCK explicit locking hints from SQL Server [71] inside a Read Committed database transaction. As a result, applications only pay the performance cost of ensuring consistency for specific operations, and developers potentially have less mental burden as fewer ad hoc constructs are involved. However, not all ad hoc transactions can benefit from these coordination hints, e.g., OCC primitives are absent. Meanwhile, database systems usually support only a subset of the listed hints, and for the same type of hints, they might exhibit different semantics (Table 9). For example, in MySQL, if any table is explicitly locked, accesses to non-explicitly-locked tables are denied [83, Section 13.3.6]; other database systems do not have this restriction. Furthermore, the tight coupling of ad hoc transactions and business logic makes migration nontrivial. In short, existing database systems have provided some but not all necessary utilities to address application demands embodied in ad hoc transactions. Thus, we believe that new abstractions and tools are needed. Below we discuss a few.

OCC Primitives. The CC of existing major database systems is based on either 2PL or **multiversion concurrency control (MVCC)**[112, Part 9]. As a result, if the application requires OCC, e.g., to deal with multi-request interactions (Section 3.1.2), developers have to craft optimistic ad hoc transactions. Therefore, we believe new OCC primitives are required and, given that many systems are closed-source, they should be provided at the ORM layer. One possible format is an *optimistic transaction declaration*, @OptimisticallyTransactional. Instead of fully delegating the coordination to database transactions, ORMs are responsible for internally tracking read/write sets of each declared optimistic transaction and atomically validating and committing changes. Another proposal is *continuation for optimistic transactions*: save(*trans*)→*tid* and restore(*tid*)→*trans*, which aid in handling multi-request interactions. Having ORMs offering boilerplate procedures reduces application complexity and the chance of errors. Meanwhile, the semantics captured by new interfaces open up opportunities for further optimization.

Proxy Module for Existing Hints. To expose advanced functionalities of existing database systems while hiding their differences, we argue for an application-level proxy module that provides general coordination customization interfaces. This module could be integrated into the ORM system or presented as a standalone system. For generality, this module should provide fallbacks when the database system in use does not support certain hints. For example, the module should provide a database table–based lock implementation as the fallback of explicit user locks.

Development Support Tools. To help improve existing, highly complex applications coupled with ad hoc transactions, we believe new development support tools must be devised to help developers locate ad hoc transactions, identify potential correctness and performance issues, and fix them by providing reliable suggestions. Ultimately, such tools should transform most ad hoc transactions into more modular forms, either database transactions or the new abstraction mentioned above.

7 RELATED WORK

Understanding Synchronization in Real Applications. Several studies have investigated how applications use manual coordination methods to deal with concurrency. A previous study [116] has identified the phenomenon of ad hoc transactions and investigated their characteristics, correctness, and performance. The extension to the previous study in this work is four-fold. First, we clarified the concept of ad hoc transactions through a detailed comparison with database transactions, which examined both conceptual details and semantics. Second, we thoroughly investigated on failure handling of ad hoc transactions, including how developers have (or have not) handled various types of failures and how actual failures affect these applications, which is only touched on the surface by the previous study. Third, we carefully analyzed the motivation behind ad hoc transactions, which was previously left as an open question, and discussed its implication as well as potential avenues for future research. Finally, we presented additional details of how ad hoc transactions are constructed, how their errors manifest, and how they perform regarding latency and scalability.

In addition, Bailis et al. [9] studied the use of ORM's invariant validation APIs to ensure application integrity, while Warszawski and Bailis [119] focused on using database transactions by web applications. We have discussed and compared with these works in depth in Section 2.2. Cheng et al. [20] have conducted an extensive study on concurrency-related issues of real-world open-source database-backed applications, focusing on the root causes, consequences, and remedies. One of their interesting findings resonating with ours is that instead of upgrading to Serializable database transactions, developers often resort to ad hoc solutions such as locks to mitigate such bugs, primarily due to performance concerns. Meanwhile, Xiong et al. [123] surveyed another type of manual coordination—ad hoc loops over synchronization variables in multi-threaded programs. Unlike (ad hoc) transactions, ad hoc loops provide low-level mutual exclusion to help programs safely access shared in-memory variables instead of transactional isolation for accessing external databases. Despite the differences with ad hoc transactions, Xiong et al. have found that ad hoc loops can also have diverse implementations and are prone to correctness issues.

Enhancing Concurrency Control with Application Semantics. Application semantics could be used to improve the parallelism of transaction processing and reduce aborts [16, 38]. Operation commutativity is one such helpful property, which allows parallel execution of execute commutative writes without blocking the initiating transactions [60, 78, 120]. Badrinath and Ramamritham [7] have discovered that certain non-commutative operations can also be executed in parallel if they are recoverable, which is determined by the semantics of the data type of the operands. These methods could potentially alleviate the need for ad hoc transactions to some extent, such as those performing partial coordination (Section 3.1.1), if they were implemented in existing database systems. However, realizing the benefits requires non-trivial manual efforts: one needs to overhaul the database systems to incorporate those data type– and operation-specific optimizations, and rewrite the applications to use the new interfaces. Given the popularity of ORMs among web applications, one might need to further enhance ORMs to provide proper support, which comes with challenges like finding the right level of abstraction.

The Escrow method and variants [45, 82] target hotspot updates to aggregate fields, such as account balances and stock quantities. Instead of locking the field until transaction commit, the database system handles Escrow updates by maintaining a range of potential values for each field that reflects the potential outcomes of all pending updates. With uncertain field values, the database system accepts additional precondition predicates for Escrow updates, which are tested against value ranges upon receiving updates and maintained for the duration of the initiating transaction. Such usage scenarios are commonly seen in ad hoc transactions (e.g., Figures 1(b), 2, and 5) and could benefit from making these if they were implemented in existing database systems. Similar to the previous methods, manual efforts in adapting the database systems, the ORMs, and the applications are required to realize the benefits.

Transaction repair [25, 122] is a technique that uses re-execution to avoid abort upon conflicts, and some ad hoc transactions have embodied this idea in their coordination. The key idea is to extract dependencies in the submitted transaction to determine the minimal set of operations that require re-execution using the latest data. Therefore, these methods require analyzing transaction logic expressed as stored procedures before execution. However, web applications submit transactions interactively instead of stored procedures, keeping computation logic and dependencies outside the database system. Therefore, besides manual efforts, the challenging task of applying program analysis over complex web applications also hinders the systematic adoption of this technique.

Ensuring Application Correctness with Relaxed Consistency. Several protocols have been proposed to provide applications with transactional capabilities over non-transactional backend storage systems. These protocols can alleviate developers from the burden of writing ad hoc transactions that perform cross–data store transactions. Dey et al. [27, 28] propose an application-level protocol, Cherry Garcia, which provides ACID Snapshot Isolation transactions composed of KV operations over heterogeneous KV stores, such as Azure Storage and Google Cloud Storage. Epoxy [59] is a recent proposal that also provides Snapshot Isolation ACID transactions over heterogeneous data stores, and it supports a wider range of operations, including searches or aggregations.

Another line of research has focused on ensuring application correctness under relaxed consistency guarantees, typically under a distributed and replicated setting. Although the applications and ad hoc transactions that we studied are not distributed, these ideas that aim to avoid manual coordination are still relevant. RedBlue consistency [66] executes some operations (a.k.a. red operations) with linearizability and others (a.k.a. blue operations) with causal consistency so that applications pay the coordination cost only when necessary. Li et al. [65] proposed the SIEVE framework to automatically generate red and blue operations based on application code and user-specified invariants. Conit [128, 129] consistency allows applications to specify the degree of acceptable inconsistency measured by numerical error, order error, and staleness. The Demarcation protocol [13, 14] and Homeostasis protocol [107] both allow distributed transactions to execute locally with stale snapshots of remote data items as long as the staleness does not violate correctness. The former requires explicitly specifying linear consistency constraints to decide when local execution is feasible. In contrast, the latter uses program analysis to infer such constraints automatically and supports more complex constraints. Balegas et al. [10] propose to preserve application invariants by introducing compensation updates to transparently correct inconsistency caused by weakly consistent replication. Balegas et al. [11] propose Explicit Consistency, which strengthens eventual consistency by ensuring specified application invariants during concurrent execution. They statically analyze application logic to find unsafe operations and remedy them using either reservation [82, 85, 111] or **conflict-free replicated data types (CRDTs)** [110]. Bailis et al. [8] introduce invariant confluence, a property that states whether a set of transactions can be executed without coordination while preserving given application invariants, and an analysis to determine this property. Alvaro et al. [3] propose an order-insensitive programming language, Bloom, which encourages eliminating ordering requirements over concurrent events so that application consistency is respected without coordination [54].

Improving Performance of Database-Backed Applications. Advanced locking methods help reduce false conflicts. Data association–aware locking methods [40, 63] have been proposed for **object-oriented database management systems (OODBMSs)** [4, 5, 12], which are similar to those in ad hoc transactions (Section 3.3.1). In OODBMSs, objects are naturally accessed via association relationships, enabling the database to provide this optimization natively. Whereas in web applications, ORM frameworks hide this access pattern, and developers have to write this optimization manually. To reduce false conflicts of gap locks, Graefe [42] proposed a method that combines ghost records (i.e., logically deleted records) with hierarchical locking [46]. This method splits index intervals when they are larger than requested key ranges, eliminating false conflicts when the original query predicate contains only equality or range conditions, such as the second example in Section 3.3.2.

Meanwhile, many analysis methods are derived for database-backed applications to identify performance issues. To avoid deadlocks in web applications, Grechanik et al. [47, 48] proposed a method that combines runtime monitoring and offline hold-and-wait cycle detection. Their methods require the knowledge of the application's outbound SQL statements, while in web applications, most SQL statements are generated at runtime using ORMs. To detect deadlocks in ORM-based applications, Dong et al. [32] proposed to analyze applications using concolic execution techniques to achieve better generality and precision. Researchers have also studied performance issues caused by ORMs [125, 126] and proposed tools to fix them automatically [19, 127].

8 CONCLUSION

This paper presents the first comprehensive study of real-world ad hoc transactions. We examined 91 cases from eight popular open-source web applications and identified the pervasiveness and importance of ad hoc transactions. We showed that ad hoc transactions are much more flexible than database transactions, which is a double-edged sword—they potentially have performance benefits but are prone to correctness issues.

REFERENCES

[1] A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph. D. Dissertation. Massachusetts Institute of Technology, USA.

[2] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering* (San Diego, CA, USA) *(ICDE'00)*. IEEE Computer Society, Washington, DC, USA, 67–78. https://doi.org/10.1109/ICDE.2000.839388

[3] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency analysis in bloom: A calm and collected approach. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research* (Asilomar, CA, USA) *(CIDR'11)*. 249–260. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

[4] Timothy Andrews and Craig Harris. 1987. Combining language and database advances in an object-oriented development environment. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Orlando, Florida, USA) *(OOPSLA'87)*. Association for Computing Machinery, New York, NY, USA, 430–440. https://doi.org/10.1145/38765.38847

[5] Malcolm Atkinson, David DeWitt, David Maier, François Bancilhon, Klaus Dittrich, and Stanley Zdonik. 1990. The object-oriented database system manifesto. In *Deductive and Object-Oriented Databases*, Won Kim, Jean-Marie Nicolas, and Shojiro Nishio (Eds.). North-Holland, Amsterdam, 223–240. https://doi.org/10.1016/B978-0-444-88433-6.50020-4

[6] Atomikos 2023. *Atomikos ExtremeTransactions.* Retrieved March 24, 2023 from https://www.atomikos.com/Main/ExtremeTransactions

[7] B. R. Badrinath and Krithi Ramamritham. 1992. Semantics-based concurrency control: Beyond commutativity. *ACM Trans. Database Syst.* 17, 1 (March 1992), 163–199. https://doi.org/10.1145/128765.128771

[8] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[9] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD'15)*. Association for Computing Machinery, New York, NY, USA, 1327–1342. https://doi.org/10.1145/2723372.2737784

[10] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. 2018. IPA: Invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 404–418. https://doi.org/10.14778/3297753.3297760

[11] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys'15)*. Association for Computing Machinery, New York, NY, USA, Article 6, 16 pages. https://doi.org/10.1145/2741948.2741972

[12] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, and Hyoung-Joo Kim. 1987. Data model issues for object-oriented applications. *ACM Trans. Inf. Syst.* 5, 1 (Jan. 1987), 3–26. https://doi.org/10.1145/22890.22945

[13] Daniel Barbará and Hector Garcia-Molina. 1992. The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the 3rd International Conference on Extending Database Technology: Advances in Database Technology (EDBT'92)*. Springer-Verlag, Berlin, 373–388.

[14] Daniel Barbará-Millá and Hector Garcia-Molina. 1994. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal* 3, 3 (Jul. 1994), 325–353. https://doi.org/10.1007/BF01232643

[15] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD'95)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

[16] A. J. Bernstein, P. M. Lewis, and S. Lu. 2000. Semantic conditions for correctness at different isolation levels. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'00)*. IEEE Computer Society, USA, 57.

[17] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[18] Broadleaf Commerce. 2021. *BroadleafCommerce*. https://github.com/BroadleafCommerce/BroadleafCommerce

[19] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1001–1012. https://doi.org/10.1145/2568225.2568259

[20] Chaoyi Cheng, Mingzhe Han, Nuo Xu, Spyros Blanas, Michael D. Bond, and Yang Wang. 2023. Developer's responsibility or database's responsibility? Rethinking concurrency control in databases. In *Proceedings of the 13th Biennial Conference on Innovative Data Systems Research* (Amsterdam, The Netherlands) *(CIDR'23)*. https://www.cidrdb.org/cidr2023/papers/p30-cheng.pdf

[21] Civilized Discourse Construction Kit. 2021. *Discourse*. https://github.com/discourse/discourse

[22] Civilized Discourse Construction Kit. 2021. *MiniSql*. https://github.com/discourse/mini_sql

[23] Claire. 2019. *Avoid Race Condition when Streaming Deleted Statuses*. https://github.com/mastodon/mastodon/pull/10280

[24] danielcolgrove. 2021. *Comments on Issue 2472*. https://github.com/BroadleafCommerce/BroadleafCommerce/issues/2472#issuecomment-762425223

[25] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2017. Transaction repair for multi-version concurrency control. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD'17)*. Association for Computing Machinery, New York, NY, USA, 235–250. https://doi.org/10.1145/3035918.3035919

[26] DB-Engines. 2021. *DB-Engines Ranking of Relational DBMS*. Retrieved Dec. 15, 2021 from https://db-engines.com/en/ranking/relational+dbms

[27] Akon Dey, Alan Fekete, and Uwe Röhm. 2013. Scalable transactions across heterogeneous NoSQL key-value data stores. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1434–1439. https://doi.org/10.14778/2536274.2536331

[28] Akon Dey, Alan Fekete, and Uwe Röhm. 2015. Scalable distributed transactions across heterogeneous stores. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering Workshops*. (Seoul, Korea) *(ICDE'15)*. IEEE Computer Society, Washington, DC, USA, 125–136. https://doi.org/10.1109/ICDE.2015.7113278

[29] Diaspora Inc. 2021. *diaspora**. https://github.com/diaspora/diaspora

[30] Django Software Foundation. 2021. *Django*. https://www.djangoproject.com

[31] Zhiyuan Dong, Chuzhe Tang, Jiachen Wang, Zhaoguo Wang, Haibo Chen, and Binyu Zang. 2020. Optimistic transaction processing in deterministic database. *Journal of Computer Science and Technology* 35, 2 (March 2020), 382–394. https://doi.org/10.1007/s11390-020-9700-5

[32] Zhiyuan Dong, Zhaoguo Wang, Chuanwei Yi, Xian Xu, Jinyuan Zhang, Jinyang Li, and Haibo Chen. 2023. Database deadlock diagnosis for large-scale ORM-based web applications. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2864–2877. https://doi.org/10.1109/ICDE55515.2023.00219

[33] Double Chain Tech. 2021. *SCM Biz Suite*. https://github.com/doublechaintech/scm-biz-suite

[34] Elastic. 2023. *Elasticsearch*. Retrieved March 24, 2023 from https://github.com/elastic/elasticsearch

[35] Elastic. 2023. *Seata: Simple Extensible Autonomous Transaction Architecture*. Retrieved March 24, 2023 from https://github.com/seata/seata

[36] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (Nov. 1976), 624–633. https://doi.org/10.1145/360363.360369

[37] FIT2CLOUD. 2021. *JumpServer*. https://github.com/jumpserver/jumpserver

[38] Hector Garcia-Molina. 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.* 8, 2 (June 1983), 186–213. https://doi.org/10.1145/319983.319985

[39] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *SIGMOD Rec.* 16, 3 (Dec. 1987), 249–259. https://doi.org/10.1145/38714.38742

[40] Jorge F. Garza and Won Kim. 1988. Transaction management in an object-oriented database system. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD'88)*. Association for Computing Machinery, New York, NY, USA, 37–45. https://doi.org/10.1145/50202.50206

[41] GitLab B.V. 2021. *GitLab*. https://gitlab.com/gitlab-org/gitlab-foss/

[42] Goetz Graefe. 2007. Hierarchical locking in B-tree indexes. In *Proceedings of The BTW 2007 Conference on Database Systems in Business,Technology and Web (BTW'07)*, Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus (Eds.). Gesellschaft für Informatik e. V., Bonn, 18–42. https://dl.gi.de/20.500.12116/31818

[43] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD'13)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2463676.2465325

[44] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The recovery manager of the system R database manager. *ACM Comput. Surv.* 13, 2 (June 1981), 223–242. https://doi.org/10.1145/356842.356847

[45] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[46] J. N. Gray, R. A. Lorie, and G. R. Putzolu. 1975. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases* (Framingham, Massachusetts) *(VLDB'75)*. Association for Computing Machinery, New York, NY, USA, 428–451. https://doi.org/10.1145/1282480.1282513

[47] Mark Grechanik, B. M. Mainul Hossain, and Ugo Buy. 2013. Testing database-centric applications for causes of database deadlocks. In *Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation* (Luxembourg, Luxembourg) *(ICST'13)*. IEEE Computer Society, Washington, DC, USA, 174–183. https://doi.org/10.1109/ICST.2013.19

[48] Mark Grechanik, B. M. Mainul Hossain, Ugo Buy, and Haisheng Wang. 2013. Preventing database deadlocks in applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 356–366. https://doi.org/10.1145/2491411.2491412

[49] The PostgreSQL Global Development Group. 2021. *PostgreSQL 13.3 Documentation*.

[50] H2 Database Engine. 2019. *Prepared XA transaction lost?* https://github.com/h2database/h2database/issues/2347

[51] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* 15, 4 (Dec. 1983), 287–317. https://doi.org/10.1145/289.291

[52] Pat Helland. 2007. Life beyond distributed transactions: An apostate's opinion. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research* (Asilomar, CA, USA) *(CIDR'07)*. 132–141. http://cidrdb.org/cidr2007/papers/cidr07p15.pdf

[53] Pat Helland. 2016. Life beyond distributed transactions: An apostate's opinion. *Queue* 14, 5 (Oct. 2016), 69–98. https://doi.org/10.1145/3012426.3025012

[54] Joseph M. Hellerstein and Peter Alvaro. 2020. Keeping CALM: When distributed consistency is easy. *Commun. ACM* 63, 9 (Aug. 2020), 72–81. https://doi.org/10.1145/3369736

[55] M. Herlihy. 1990. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.* 15, 1 (March 1990), 96–124. https://doi.org/10.1145/77643.77647

[56] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 629–642. https://doi.org/10.14778/3377369.3377373

[57] Iteration Labs. 2021. *Huginn*. https://github.com/refinery/refinerycms

[58] J. R. Jordan, J. Banerjee, and R. B. Batman. 1981. Precision locks. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data* (Ann Arbor, Michigan) *(SIGMOD'81)*. Association for Computing Machinery, New York, NY, USA, 143–147. https://doi.org/10.1145/582318.582340

[59] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID transactions across diverse data stores. *Proc. VLDB Endow.* 16, 11 (Aug. 2023), 2742–2754. https://doi.org/10.14778/3611479.3611484

[60] Akhil Kumar and Michael Stonebraker. 1988. Semantics based transaction management techniques for replicated data. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD'88)*. Association for Computing Machinery, New York, NY, USA, 117–125. https://doi.org/10.1145/50202.50215

[61] H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[62] Jean-Philippe Lang. 2021. *Redmine*. https://github.com/redmine/redmine

[63] Suh-Yin Lee and Ruey-Long Liou. 1996. A multi-granularity locking model for concurrency control in object-oriented database systems. *IEEE Trans. Knowl. Data Eng.* 8, 1 (1996), 144–156. https://doi.org/10.1109/69.485643

[64] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. Multi-version range concurrency control in deuteronomy. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2146–2157. https://doi.org/10.14778/2831360.2831368

[65] Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Annual Technical Conference* (Philadelphia, PA, USA) *(USENIX ATC'14)*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, USA, 281–292.

[66] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 265–278.

[67] X/Open Company Limited. 1991. Distributed Transaction Processing: The XA Specification.

[68] David B. Lomet. 2004. Simple, robust and highly concurrent b-trees with node deletion. In *Proceedings of the 20th International Conference on Data Engineering* (Boston, MA, USA) *(ICDE'04)*, Z. Meral Özsoyoglu and Stanley B. Zdonik (Eds.). IEEE Computer Society, Washington, DC, USA, 18–27. https://doi.org/10.1109/ICDE.2004.1319981

[69] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys'12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[70] Microsoft 2017. *Designing Transactional Web Applications*. Retrieved March 26, 2023 from https://learn.microsoft.com/en-us/previous-versions/iis/6.0-sdk/ms524850(v=vs.90)

[71] Microsoft. 2020. *Table Hints (Transact-SQL)*. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table?view=sql-server-ver15

[72] Microsoft. 2021. *Remarks on SET TRANSACTION ISOLATION LEVEL (Transact-SQL)*. Microsoft. https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver15#remarks

[73] Vlad Mihalcea. 2015. *High-Performance Java Persistence*. Vlad Mihalcea, Romania.

[74] Vlad Mihalcea, Steve Ebersole, Andrea Boriero, Gunnar Morling, Gail Badner, Chris Cranford, Emmanuel Bernard, Sanne Grinovero, Brett Meyer, Hardy Ferentschik, Gavin King, Christian Bauer, Max Rydahl Andersen, Karel Maesen, Radim Vansa, and Louis Jacomet. 2021. *Hibernate ORM 5.5.7.Final User Guide*. Red Hat. https://docs.jboss.org/hibernate/orm/5.5/userguide/html_single/Hibernate_User_Guide.html

[75] C. Mohan. 1990. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 392–405.

[76] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.* 17, 1 (Mar. 1992), 94–162. https://doi.org/10.1145/128765.128770

[77] C. Mohan and Frank Levine. 1992. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD'92)*. Association for Computing Machinery, New York, NY, USA, 371–380. https://doi.org/10.1145/130283.130338

[78] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 511–524.

[79] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD'15)*. Association for Computing Machinery, New York, NY, USA, 677–689. https://doi.org/10.1145/2723372.2749436

[80] OASIS Web Services Transaction (WS-TX) TC. 2009. *Web Services Atomic Transaction (WS-AtomicTransaction)*. OASIS. https://docs.oasis-open.org/ws-tx/wsat/2006/06

[81] OASIS Web Services Transaction (WS-TX) TC. 2009. *Web Services Coordination (WS-Coordination)*. OASIS. https://docs.oasis-open.org/ws-tx/wscoor/2006/06

[82] Patrick E. O'Neil. 1986. The escrow transactional method. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986), 405–430. https://doi.org/10.1145/7239.7265

[83] Oracle. 2020. *MySQL 8.0 Reference Manual (Including MySQL NDB Cluster 8.0)*.

[84] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 433–448.

[85] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. 2003. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services* (San Francisco, California) *(MobiSys'03)*. Association for Computing Machinery, New York, NY, USA, 43–56. https://doi.org/10.1145/1066116.1189038

[86] Dan Pritchett. 2008. BASE: An acid alternative. *Queue* 6, 3 (may 2008), 48–55. https://doi.org/10.1145/1394127.1394128

[87] Project Concerto. 2021. *API Controller did not Implement Order Version Check*. https://github.com/spree/spree/issues/10748

[88] Project Concerto. 2021. *Crash while Processing Payments Leads to Unexpected Behavior*. https://github.com/spree/spree/issues/10744

[89] Project Concerto. 2021. *Implementation Issue in Order Lock*. https://github.com/spree/spree/issues/10697

[90] Project Concerto. 2021. *Mixing Active Record & Mini_sql Leads to Unexpected Behavior*. https://meta.discourse.org/t/mixing-active-record-mini-sql-leads-to-unexpected-behavior/176504

[91] Project Concerto. 2021. *A More Efficient Redis Lock*. https://meta.discourse.org/t/a-more-efficient-redis-lock/177841

[92] Project Concerto. 2021. *Possible Race Condition when Paying with Store Credit*. https://github.com/spree/spree/issues/11284

[93] Project Concerto. 2021. *Race Condition in Downsize_Upload Script*. https://meta.discourse.org/t/race-condition-in-downsize-upload-script/177152

[94] Project Concerto. 2021. *Redis lock's TTL May Leads to Potential Bugs*. https://github.com/tootsuite/mastodon/issues/15645

[95] Project Concerto. 2021. *Session Order Lock may be Discarded Unexpectedly*. https://github.com/BroadleafCommerce/BroadleafCommerce/issues/2555

[96] Project Concerto. 2021. *Sku Inconsistent Caused by Concurrent Checkout*. https://github.com/BroadleafCommerce/BroadleafCommerce/issues/2472

[97] Calton Pu, Gail E. Kaiser, and Norman C. Hutchinson. 1988. Split-transactions for open-ended activities. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB'88)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 26–37.

[98] Rails Core Team. 2021. *Active Record*. https://guides.rubyonrails.org/active_record_basics.html

[99] Rails Core Team. 2021. *Ruby on Rails*. https://rubyonrails.org

[100] Red Hat. 2021. *Hibernate ORM*. https://hibernate.org/orm/

[101] Manuel Reimer. 1983. Solving the phantom problem by predicative optimistic concurrency control. In *Proceedings of the 9th International Conference on Very Large Data Bases (VLDB'83)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 81–88.

[102] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design principles for scaling multi-core OLTP under high contention. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD'16)*. Association for Computing Machinery, New York, NY, USA, 1583–1598. https://doi.org/10.1145/2882903.2882958

[103] Willem Renzema. 2016. *Willem Renzema's Answer to "How to Properly use Transactions and Locks to Ensure Database Integrity?"*. Retrieved March 26, 2023 from https://stackoverflow.com/a/40773638

[104] Resolve Digital. 2021. *Refinery CMS*. https://github.com/refinery/refinerycms

[105] Eugen Rochko. 2019. *Comments on Commit a359be23*. https://github.com/mastodon/mastodon/pull/4642/commits/\a359be23a3c26fe7a824ce9bb0f1faaaac3d02b6#r134231414

[106] Eugen Rochko and other Mastodon contributors. 2021. *Mastodon*. https://github.com/mastodon/mastodon

[107] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. 2015. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD'15)*. Association for Computing Machinery, New York, NY, USA, 1311–1326. https://doi.org/10.1145/2723372.2723720

[108] Saleor Commerce. 2021. *Saleor Commerce*. https://github.com/saleor/saleor

[109] Seata 2023. *Application Model and Scenario Analysis for TCC Mode (in Chinese)*. Retrieved March 26, 2023 from http://seata.io/zh-cn/blog/tcc-mode-applicable-scenario-analysis.html

[110] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin, Berlin, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[111] Liuba Shrira, Hong Tian, and Doug Terry. 2008. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware* (Leuven, Belgium) *(Middleware'08)*. Springer-Verlag, Berlin, 42–61.

[112] A. Silberschatz, H. F. Korth, and S. Sudarshan. 2011. *Database System Concepts*. McGraw-Hill, New York, NY, USA.

[113] Spree Commerce. 2021. *Spree Commerce*. https://spreecommerce.org

[114] Spring 2023. *Spring Framework Data Access*. Retrieved March 24, 2023 from https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html

[115] Stripe 2023. *Stripe, Payment Processing Platform for the Internet*. Retrieved March 24, 2023 from https://stripe.com

[116] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad hoc transactions in web applications: The good, the bad, and the ugly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD'22)*. Association for Computing Machinery, New York, NY, USA, 4–18. https://doi.org/10.1145/3514221.3526120

[117] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania) *(SOSP'13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[118] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-performance transactions via learned concurrency control. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 198–216. https://www.usenix.org/conference/osdi21/presentation/wang-jiachen

[119] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD'17)*. Association for Computing Machinery, New York, NY, USA, 5–20. https://doi.org/10.1145/3035918.3064037

[120] W. E. Weihl. 1988. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.* 37, 12 (Dec. 1988), 1488–1505. https://doi.org/10.1109/12.9728

[121] Wentang. 2022. *How do Cloud Users use Cloud Database Systems? (in Chinese)*. Retrieved March 24, 2023 from https://zhuanlan.zhihu.com/p/525648427

[122] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. 2016. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD'16)*. Association for Computing Machinery, New York, NY, USA, 1689–1704. https://doi.org/10.1145/2882903.2915202

[123] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 163–176. https://www.usenix.org/conference/osdi10/ad-hoc-synchronization-considered-harmful

[124] X/Open. 1991. *Distributed Transaction Processing: The XA Specification.*

[125] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (Singapore, Singapore) *(CIKM'17)*. Association for Computing Machinery, New York, NY, USA, 1299–1308. https://doi.org/10.1145/3132847.3132954

[126] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How not to structure your database-backed web applications: A study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE'18)*. Association for Computing Machinery, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[127] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. PowerStation: Automatically detecting and fixing inefficiencies of database-backed web applications in IDE. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*

(Lake Buena Vista, FL, USA) *(ESEC/FSE'18)*. Association for Computing Machinery, New York, NY, USA, 884–887. https://doi.org/10.1145/3236024.3264589

[128] Haifeng Yu and Amin Vahdat. 2000. Design and evaluation of a continuous consistency Model for Replicated Services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4* (San Diego, California) *(OSDI'00)*. USENIX Association, USA, Article 21.

[129] Haifeng Yu and Amin Vahdat. 2002. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.* 20, 3 (Aug. 2002), 239–282. https://doi.org/10.1145/566340.566342

[130] Xiaodong Zhang. 2021. The Synchronized used to Prevent Concurrency Doesn't Work as Expected (in Chinese). https://github.com/doublechaintech/scm-biz-suite/issues/17

Received 14 April 2023; revised 3 September 2023; accepted 5 December 2023