Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly

Chuzhe Tang[1,2], Zhaoguo Wang[1,2], Xiaodong Zhang[1,2], Qianmian Yu[1,2]
Binyu Zang[1,2], Haibing Guan[3], Haibo Chen[1,2]
[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[2]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
[3]Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University

ABSTRACT

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 8 popular open-source web applications, we find 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 to improve performance in contentious workloads by utilizing application semantics such as access patterns. Based on the findings, we discuss the implications of ad hoc transactions to the database research community.

CCS CONCEPTS

• **Information systems → Database transaction processing**; **Web applications**.

KEYWORDS

database transactions, ad hoc transactions, web applications

ACM Reference Format:
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 (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526120

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 [69]). 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 [4, 5, 16, 21, 25, 42, 43, 53, 68, 82, 84].

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 [15].

Figure 1 shows three real-world examples of ad hoc transactions from open-source web applications, Broadleaf [10], Mastodon [75], and Discourse [12]. 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 8 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 [71], Active Record [69], and Django [20]). 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 3 e-commerce applications. 31 ad hoc

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06...$15.00
https://doi.org/10.1145/3514221.3526120

```
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.

Figure 1: Ad hoc transaction examples. Coordinated DB accesses are shaded yellow; ad hoc constructs are shaded green.

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, 58 cases use a single, fine-grained lock to coordinate multiple database operations. At first glance, we suspected that these cases have missed necessary coordination and are thus incorrect. However, after checking these cases in-depth, we found that not all operations require coordination. One reason is that some objects are always associatively accessed so that a single lock is sufficient for correctness.

(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. While this large percentage might seem unsurprising considering the variety of 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 [71] and Active Record [69]. 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*.

	Feral CC *Bailis et al.* [5]	ACIDRain *Warszawski and Bailis* [83]	This work
Target	ORMs' invariant validation APIs	Database transactions	Ad hoc transactions
Aspects	1. Characteristics 2. Correctness	Correctness	1. Characteristics 2. Correctness 3. Performance
Issue types	Insufficient isolation	1. Insufficient isolation 2. Incorrect trans. scope	1. Incorrect sync. primitives 2. Incorrect ad hoc trans. scope 3. Incorrect failure handling

Table 1: Comparison with Feral CC and ACIDRain.

ORM frameworks usually allow developers to use database transactions explicitly, with interfaces that directly translate to Transaction Start, Commit, and Abort statements. Developers 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 [83].

Besides database transactions, ORMs also provide built-in invariant validation APIs. For example, Active Record [69] 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 CC 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. [5] studied "feral" CC—ORM's invariant validation APIs, and Warszawski and Bailis [83] 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.

Application	Category	Language/ORM	RDBMS	Stars	Contri-butors
Discourse [12]	Forum	Ruby/Active Record	PG	33.8k	776
Mastodon [75]	Social network	Ruby/Active Record	PG	24.6k	644
Spree [81]	E-commerce	Ruby/Active Record	PG, MY	11.4k	855
Redmine [39]	Project mgmt.	Ruby/Active Record	PG, MY, +	4.2k	8
Broadleaf [10]	E-commerce	Java/Hibernate	PG, MY, +	1.5k	73
SCM Suite [22]	Supply chain	Java/Hibernate	PG, MY	1.5k	2
JumpServer [24]	Access control	Python/Django	PG, MY, +	16.8k	88
Saleor [77]	E-commerce	Python/Django	PG, MY, +	13.9k	181

Table 2: The applications corpus. The "RDBMS" column lists supported RDBMSs. "PG/MY/+" refers to PostgreSQL/MySQL/others.

Specifically, Bailis et al. studied how Rails [70] 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 [4], 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 (§3), correctness (§4), and performance (§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 [83, §3.2].

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. 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. 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 1a and 1b use locks to isolate conflicting operations, e.g., the concurrent reading and writing of the same cart. Similarly, Figure 1c 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 8 representative applications of 6 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 [10] is the highest star-ed Java e-commerce application on GitHub and Spree [81] is the most popular e-commerce application in Ruby. 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.

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

Table 3: Ad hoc transactions are mainly used in core APIs.

App.	Cases		CC alg.	
	Total	Buggy	Lock	Valid.
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

† SCM Suite generates source code for different suppliers from templates; only cases in templates are counted. In its (generated) demo, there are 167 cases.

Table 4: Statistics of identified ad hoc transactions. *Buggy* cases refer to those with correctness issues. All cases coordinate concurrency either with *Lock*s or OCC-style *Valid*ations.

> 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 [10], Spree [81], and Saleor [77], 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?
- Can ad hoc transactions always ensure the correct semantics?
- How is ad hoc transactions' performance, especially in comparison with database transactions?

[1] Redmine [39] is the second popular project management application now. Its popularity has waned since we picked it as the investigation target.

3 CHARACTERISTICS OF AD HOC TRANSACTIONS

We have carefully studied the 91 identified ad hoc transaction cases. 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 (2PL) and its variants commonly used by existing database systems [23, 26, 28, 37, 41, 50, 52, 73]. Unlike database transactions, pessimistic ad hoc transactions' locking primitives are usually implemented from scratch by application developers (e.g., Figures 1a and 1b) or provided by other systems (see §3.2). Meanwhile, optimistic ad hoc transactions execute operations aggressively and validate the execution result before writing updates back to the database system (Figure 1c). This approach is similar to OCC and its variants used in existing database systems [35, 36, 38, 53, 72, 82].

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 [17] 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.*

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 following example from the Spree e-commerce application [81].

```
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)
```

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 Transaction Start/Commit may worsen performance, as all the updates will be performed under the same isolation level. Consider 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 [48, §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 [57, §15.7.2.3].

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 Snapshot Isolation for Repeatable Read. 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 [9]. 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 [33, §13.2.2].

Ideally, developers should exclude these timestamp updates from the scope of database transactions or switch the isolation level with database interfaces [47]. 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

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

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. Below is an example derived from the Discourse forum application [12] 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.

```
Request 1 // fetch a post & increment view count
  in: post_id
  Update Post Set view_cnt=view_cnt+1, ver=ver+1 Where id=post.id
  post := Select * From Posts 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
```

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.

Extensions to database transactions were proposed for LLTs, such as *Sagas* [25] and *savepoints* [29, 51]. 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 Operation 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 8 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 following example simplified from the timeline feature of the Mastodon social network application [75].

```
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)
```

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 can not 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-TX [54, 55] or XA transactions [86]. However, storage systems rarely support such distributed transaction protocols, which necessitate ad hoc transactions. Dey et al. [18, 19] 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 The 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 8 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 [81], Saleor [77], and Redmine [39] use the database Select For Update statements, while SCM Suite [22] 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.

```
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
```

The above example is simplified from the Saleor e-commerce application [77], 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 [12], Mastodon [75], and JumpServer [24], have locks implemented from scratch. Interestingly, they all store lock information, including lock keys and status (locked/unlocked), in the Redis KV store. However, their implementation details are different. As shown in Figure 1b, Mastodon developers use the Redis SETNX (short for SET if Not eXists) command to insert an entry for the requested lock. Similar to the Compare and Swap (CAS) instruction, this command succeeds only if no entry with the same key exists. In contrast, Discourse developers use a combination of WATCH, GET, MULTI, and SET commands to optimistically ensure the atomicity of checking existing locks and setting new locks.[3] As a result, Discourse's Redis lock requires six additional round trips compared to Mastodon's, which only needs one [63]. Saleor 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 [10] 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: one uses a separate database table to store lock information similar to those Redis-based locks; the other two use in-memory maps for lock information. The latter two implementations differ in the specific maps used: one directly uses a concurrent map from the standard library, ConcurrentHashMap; the other uses a customized ConcurrentHashMap where developers added a least recently used (LRU) eviction policy to remove excessive lock entries. We find no clear evidence that these different implementations serve different purposes. However, we do find that different developers have introduced these implementations.

[3] WATCH adds a key, even if it does not exist in Redis yet, to a *watch set*, and MULTI detects if any change has taken place ever since for keys in the watch set.

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 1c, Active Record automatically adds version checking to the Where clause and increment version along with user-initiated updates, ensuring the atomicity between validation and commit.

When using hand-crafted validation procedures, developers must ensure the atomicity between validation and commit. As shown in the listing from §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.

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 §4 and also compare their performance in §5.

3.3 What Are The 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 [30]. 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.

Lock in ad hoc transactions could coordinate arbitrary database accesses. According to our study, 58 ad hoc transactions that use one lock to coordinate multiple database accesses. This is because the developers usually could identify the following two 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 [10], shown in Figure 1a. 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 [49, §2.11].

Read–Modify–Writes (RMWs). RMW means that a transaction first queries the data from the database system, then makes 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 [48, §7.3.3.3], which necessitate the use of Serializable. Consider the example shown in Figure 1b, in the forum application Discourse [12], RMW operations are issued when creating a new account via invitations. The invitation is first read from the RDBMS. After checking its validity, it gets updated and written back to the RDBMS. 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 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 is avoiding false conflicts. We find ad hoc transactions' fine-grained coordination are either based on columns or predicates.

Columns-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 [12], two transactions, create-post and toggle-answer, will issue the following database operations accessing the Topics table.

```
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)
```

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 RDBMS 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.[4]

Optimistic ad hoc transactions can also benefit from column-based coordination—they only need to validate whether specific column values have been updated. The following shows a more accurate representation of the edit-post transaction in Discourse [12], which we previously discussed in §3.1.2.[5]

```
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
```

[4]Nevertheless, RDBMS still executes line 6 and line 12 serially to avoid data corruption.
[5]However, the version column still exists for use in other APIs.

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, 5 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 [44, 50, 52], including MySQL and PostgreSQL. For example, in the Spree [81] e-commerce application, RDBMSs 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.

```
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)
```

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 RDBMS 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* [23, 37], 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 RDBMSs, it is more practical for them to derive a customized predicate locking scheme than for RDBMSs 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 performance and scalability of the underlying tree structures, to obtain which require significant effort [45]. 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 also need to handle failures caused by deadlocks, failed validation, database failure, and web server crashes.

Finding 5. *All pessimistic ad hoc transactions do not encounter deadlocks as they all acquire locks in the same order. Most optimistic ad hoc transactions (19 out of 26 cases) directly return an error to the user on failed validation.*

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. Each pessimistic ad hoc transaction either uses a single lock (52/65) or acquires locks in a consistent order (13/65). Thus, none of them needs to handle deadlock at runtime. As for optimistic ad hoc transactions, 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 1 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, 2 cases are equipped with manually written rollback procedures. These procedures are triggered by validation failures and will undo persisted updates.

Meanwhile, 4 cases choose to repair the inconsistent values instead of rolling back on conflicts. This idea relies on developers' knowledge of program dependency and is similar to the transaction repair optimizations [16, 84]. Consider the following example taken from the Discourse [12] forum application, a periodic background task that shrinks large images in posts.

```
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
```

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 RDBMS 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. Failures caused by crashes can be further divided into two categories: (i) database system crashes and (ii) application server crashes. When the former occurs, 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, as we previously discussed.

However, rollback statements for ongoing ad hoc transactions cannot be issued when the latter occurs. To correctly resume service after application reboot, applications need to ensure that locks acquired before crashes will not cause deadlocks, and application logic can tolerate potential intermediate database states. It is easy to avoid deadlocks: among all cases except one from Broadleaf [10], lock information does not persist—they either vanish along with crashes (in-memory locks) or expire after a given period (Redis locks). The exceptional Broadleaf case uses locks persisted in a database table. To avoid deadlock, developers associate each lock with a boot-time generated universally unique identifier (UUID) that distinguishes each boot. Thus, Broadleaf can ignore prior unreleased locks after reboot by examining the saved UUIDs.

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

4 CORRECTNESS ISSUES

The variety of implementation possibilities as we discuss in §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. The issues discussed below are surely incomplete, and we have manually verified that all issues are reproducible and cause user-noticeable consequences.

Result Summary. 69 correctness issues are found in 53 cases (Table 5a); some cases have multiple issues. Furthermore, 28 cases have severe consequences (Table 5b), such as charging customers incorrect amounts. 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[6]) 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 between validation and commit, causing correctness issues.*

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

Category	Description	Apps	Cases
Incorrect sync. primitives	Locking primitive impl./usage issues.	6	36
	Non-atomic validate-and-commit.	3	11
Incorrect ad hoc trans. 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

(a) Categorization of incorrect ad hoc transactions. Note that one ad hoc transaction can have multiple issues.

App.	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

(b) Incorrect ad hoc transactions can have severe consequences.

Table 5: Correctness issues of ad hoc transactions.

4.1.1 Locking Primitive Issues. There are 7 different lock implementations (§3.2.1) and 5 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 (§3.2.1), and both have corresponding cases of incorrect usage. Spree [81] 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 [61]. Meanwhile, SCM Suite [22] 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 [91].

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. For example, in Discourse [12], an ad hoc transaction uses locks to coordinate posts from concurrent edits. However, the post is locked after being read from the RDBMS when processing edit requests. Although the lock serializes the subsequent write-backs, the complete RMW process is not atomic, allowing one's edit to overwrite others' [76].

Incorrect Lock Implementation. 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 (§3.2.1). For lock based on Redis, Mastodon [75] gives an example where the developers implement the lease semantics. Specifically, they enable the auto-expire feature of Redis [65] 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 [14]. Furthermore, all ad hoc transactions in Mastodon are based on this incorrect lock implementation. For lock based on in-memory lock table, Broadleaf [10]'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 [66]. 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). Discourse [12] gives an example.

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

Versions are used to track changes to reviewable items (e.g., a controversial topic) and prevent conflicting administrator operations. Developers explicitly enclose the validation (line 3–5) and subsequent updates (line 6) 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 [13], 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 [62].

4.2 Incorrect Coordination Scope

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 (§3.1.1), it comes with an increased chance of leaving critical operations uncoordinated. For example, in Broadleaf [10], the ad hoc transaction that coordinates the check-out process omits coordination for all SKU-related operations. As a result, concurrent check-outs for the same SKU can lead to inconsistency between the SKU quantity decrement and the number of sold items [67]. Optimistic ad hoc transactions are subject to such errors as well. For example, in Spree [81], the transaction that decrements SKU quantity (shown in §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 [61].

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 [81], 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 RDBMS states inconsistent [59]. 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. *A small portion of issues come from complex coordination; all happen in customized abort handling (4 issues).*

Incomplete Repair. 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 [12], when updating image references of posts, developers use versions to track individual states of fetched posts from a query (pseudocode shown in §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 [64]. 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 intermediate states, the application might fail to provide normal services if crashes occur. For example, in Spree [81], a crash during check-out can leave payments in an intermediate state (i.e., having the status column equalling "processing"). Since these payments 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 being "processing" by active threads. Therefore, users can never finish the check-out [60]. 3 cases have similar issues.

5 PERFORMANCE EVALUATION

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

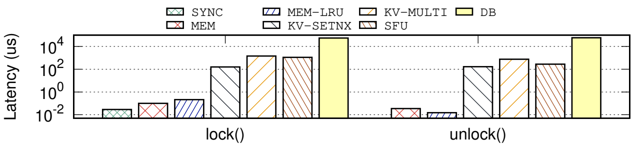

Figure 2: Latencies of different lock implementations. The invocation latency of synchronized is shown under lock().

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 × 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.[7] Figure 2 shows the results. The latency differences are of orders of magnitude. The slowest among them is the RDBMS-based one (DB), ported from Broadleaf [10], 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 [75], Discourse [12], and Saleor [77], 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 (§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.

5.2 Different Coordination Granularities

Ad hoc transactions can perform coordination at granularities rarely seen in database systems (§3.3). To understand their impact, we chose 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 measure each API's peak throughput with the original, ad hoc transaction–based codebase (denoted as AHT) and a modified one using database transactions with the weakest yet sufficient[8] isolation level instead (denoted as DBT). Table 6 lists the specific APIs, workloads, and setups. APIs used in CBC and PBC are previously described in §3.3.2; RMW's API is similar to the one described in §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.

Gran.	Application API(s)	Workload (with contention)	RDBMS	DBT isolation
RMW (§3.3.1)	check-out, Broadleaf [10]	Customers purchase the same SKU.	MySQL	Serializable
AA (§3.3.1)	like-post, Discourse [12]	Users like different posts of seven contented topics.	PostgreSQL	Serializable
CBC (§3.3.2)	create-post & toggle-answer, Discourse [12]	Assign distinct topics to user pairs, where one user creates posts and one accepts answer.	PostgreSQL	Repeatable Read
PBC (§3.3.2)	add-payment, Spree [81]	Customers submit payment options for new orders.	PostgreSQL	Serializable

Table 6: APIs and setups for evaluating coordination granularities. We obtain no-contention workloads by switching users to work with different SKUs/topics or existing orders.

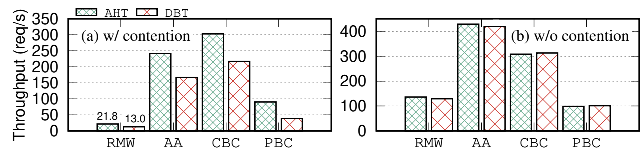

Figure 3: API throughputs using different coordination granularities.

Figure 3 shows the results. 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' noncritical 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.

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 §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

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

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

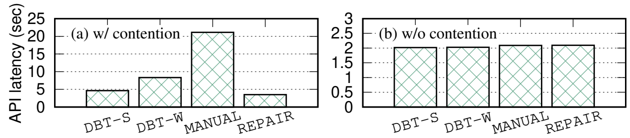

Figure 4: API Latencies using different rollback methods.

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 §3.1.2) over posts of each image, conflicting with shrink-image invocations.

Figure 4 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.

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.		

(a) Coordination hints supported by the top ten ranking RDBMSs [17]. 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.

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

† Work in conjunction with database transactions.

(b) Relationship between coordination hints and ad hoc transactions.

Table 7: Coordination hints supported by existing database systems and their relationship with ad hoc transactions.

6 DISCUSSION

We have observed that ad hoc transactions are error-prone and difficult to identify and understand, but they are still widely used in critical APIs. Thus, we believe more study is required to understand why developers use ad hoc transactions instead of other more modular approaches such as database transactions. For example, are database transactions too inefficient, inconvenient to use, or lacking critical functionalities? Different answers lead to different future database research directions. One potential answer is the lack of critical functionalities. As described in §3.1, certain coordinated business logic exposes characteristics difficult or impossible for database transactions to handle. For example, database transactions surely fall short when business procedures access multiple storage backends (§3.1.3). Developers have expressed similar concerns [90]. Another potential reason lies in the performance—we have found that ad hoc transactions could perform better than database transactions under contentious workloads and similarly under no-contention ones (§5). Likewise, developers have expressed performance concerns, e.g., they want to avoid LLTs [67, 74].

Meanwhile, many existing database systems provide interfaces for passing hints that customize the coordination. For example, PostgreSQL provides explicit user locks, where locks are identified by user-specified integers and scoped by the active session or transaction [33, §13.3.5]. However, can they help developers write ad hoc transactions or even replace them? To answer this question, we compiled a summary (Table 7a) of supported coordination hints among the top ten ranking RDBMSs [17] and found that they can in part prevent errors while retaining the benefits (Table 7b). For example, to coordinate only specific database operations (§3.1.1), we can augment them with the HOLDLOCK explicit locking hints from SQL Server [46] 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 7a). For example, in MySQL, if any table is explicitly locked, accesses to non-explicitly-locked tables are denied [57, §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)[80, Part 9]. As a result, if the application requires OCC, e.g., to deal with multi-request interactions (§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. Bailis et al. [5] studied the use of ORM's invariant validation APIs to ensure application integrity, while Warszawski and Bailis [83] focused on using database transactions by web applications. We have discussed and compared with these works in depth in §2.2. Meanwhile, Xiong et al. [85] 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.

Ensuring Correctness of Database-Backed Applications. To build applications when the underlying data store does not support transactions, Dey et al. [18, 19] propose an application-level protocol, Cherry Garcia, which provides ACID transactions with Snapshot Isolation over heterogeneous KV stores, such as Azure Storage and Google Cloud Storage. Others are concerned with applications directly operating on KV stores, especially those weakly replicated ones. Balegas et al. [6] propose to preserve application invariants by introducing compensation updates to transparently correct inconsistency caused by weakly consistent replication. Balegas et al. [7] 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 [56, 58, 79] or conflict-free replicated data types (CRDTs) [78]. Bailis et al. [4] 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. [1] propose an order-insensitive programming language, Bloom, which encourages eliminating ordering requirements over concurrent events so that application consistency is respected without coordination [34].

Improving Performance of Database-Backed Applications. The ideas embodied in ad hoc transactions' customized coordination can be found in prior research efforts. We briefly review them below.

Advanced locking methods help reduce false conflicts. Data association–aware locking methods [26, 40] have been proposed for object-oriented database management systems (OODBMSs) [2, 3, 8], which are similar to those in ad hoc transactions (§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 [27] proposed a method that combines ghost records (i.e., logically deleted records) with hierarchical locking [30]. 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 §3.3.2.

Transaction repair [16, 84] is a technique that uses re-execution to avoid abort upon conflicts. 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 RDBMS.

Meanwhile, many analysis methods are derived for database-backed applications to identify performance issues. To avoid deadlocks in web applications, Grechanik et al. [31, 32] proposed a method that combines runtime monitoring and offline hold-and-wait cycles detection. Their methods require the knowledge of outbound SQL statements of the application, while in web applications, most SQL statements are generated at runtime. Researchers have also studied performance issues caused by ORMs [87, 88] and proposed tools to fix them automatically [11, 89].

8 CONCLUSION

This paper presents the first comprehensive study of real-world ad hoc transactions. We examined 91 cases from 8 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.

ACKNOWLEDGMENTS

We appreciate the insightful discussions we had with Zhou Zhou and Jiahuan Shen at various stages of this project and the constructive feedback from the anonymous reviewers. This work is supported by the National Natural Science Foundation of China under Grant Nos. 61902242, 62132014, and 62172272 and the HighTech Support Program from Shanghai Committee of Science and Technology under Grant No. 20ZR1428100. The corresponding author is Zhaoguo Wang (zhaoguowang@sjtu.edu.cn).

REFERENCES

[1] 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

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

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

[11] 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

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

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

[14] Claire. 2019. *Avoid race condition when streaming deleted statuses.* https://github.com/mastodon/mastodon/pull/10280

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

[16] 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

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

[18] 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

[19] 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

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

[21] 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

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

[23] 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

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

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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

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

[40] 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

[41] 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

[42] 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.

[43] 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.

[44] 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

[45] 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

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

[47] 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

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

[49] 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

[50] 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.

[51] 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

[52] 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

[53] 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

[54] 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

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

[56] 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

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

[58] 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

[59] Project Concerto. 2021. *API controller did not implement order version check*. https://github.com/spree/spree/issues/10748

[60] Project Concerto. 2021. *Crash while processing payments leads to unexpected behavior*. https://github.com/spree/spree/issues/10744

[61] Project Concerto. 2021. *Implementation issue in order lock*. https://github.com/spree/spree/issues/10697

[62] 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

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

[64] Project Concerto. 2021. *Race condition in downsize_upload script*. https://meta.discourse.org/t/race-condition-in-downsize-upload-script/177152

[65] Project Concerto. 2021. *Redis lock's TTL may leads to potential bugs*. https://github.com/tootsuite/mastodon/issues/15645

[66] Project Concerto. 2021. *Session order lock may be discarded unexpectedly*. https://github.com/BroadleafCommerce/BroadleafCommerce/issues/2555

[67] Project Concerto. 2021. *Sku inconsistent caused by concurrent checkout*. https://github.com/BroadleafCommerce/BroadleafCommerce/issues/2472

[68] 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.

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

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

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

[72] 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.

[73] 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

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

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

[76] Sam Saffron. 2015. *A reply on Edit of rebaked post doesn't show in HTML; only in raw*. https://meta.discourse.org/t/edit-of-rebaked-post-doesnt-show-in-html-only-in-raw/33815/6

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

[78] 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 Heidelberg, Berlin, Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[79] 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, Heidelberg, 42–61.

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

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

[82] 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* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[83] 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

[84] 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

[85] 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

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

[87] 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

[88] 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

[89] 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

[90] Philip Z. 2021. *Comments on issue 17 (in Chinese)*. https://github.com/doublechaintech/scm-biz-suite/issues/17

[91] 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