Ad Hoc Transactions: What They Are and Why We Should Care

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
zhaoguowang@sjtu.edu.cn

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. Finally, implications of ad hoc transactions to the database research community are discussed.

1. INTRODUCTION

Today, web applications often use database systems to store and serve large amounts of data, making coordinating concurrent database operations necessary for application correctness. One common approach is using database transactions. Database transactions isolate concurrent operations by encapsulating them into individual units of work. Another widely adopted approach is using object-relational mapping (ORM) invariant validation APIs. These APIs allow developers to explicitly specify invariants, such as the uniqueness of column values, in the application code; at runtime, ORM frameworks report errors on invariant violations. So far, much work has been done to investigate and improve these two approaches [2, 16, 3, 10, 9, 6, 12, 11, 14, 1, 4].

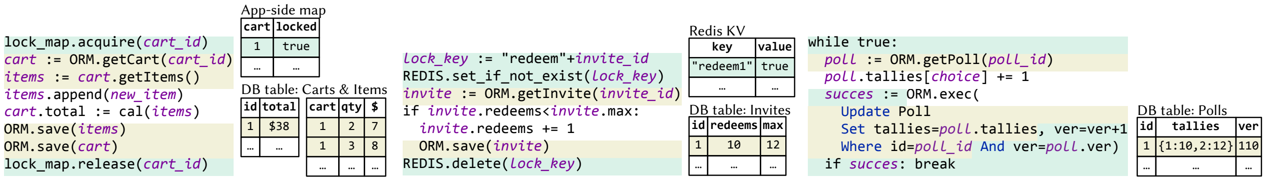
However, besides these approaches, application developers are also accustomed to coordinating critical database operations ad hoc. Specifically, developers might explicitly use locking primitives and validation procedures to implement concurrency control (CC), e.g., optimistic concurrency control (OCC), 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.

Figure 1 shows three real-world examples of ad hoc transactions from open-source web applications. In each example, the application code issues database operations via ORM frameworks 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. We briefly elaborate Figure 1a, which shows how an e-commerce application processes an add-to-cart request. First, a lock identified by the target cart ID is acquired and held unreleased until the whole business logic finishes. Then, the target cart and items in it are transparently loaded from the database system and converted into runtime objects cart and items by the ORM. Next, the new item is added to the items collection, and the total price is recalculated. Finally, the ORM persists cart and items updates to the database system. 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, and 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 belonging to six different categories. These applications are considered the most popular in their respective categories, as measured by GitHub stars. They are developed in different languages (Java, Ruby, or Python) and different ORM frameworks (Hibernate, Active Record, and Django). Our study aims to understand the characteristics of ad hoc transactions in existing web applications and their implications. This paper presents a short summary of our findings and insights, condensed from the conference version [13]. Briefly, we discovered the following interesting, alarming, and perceptive findings.

(i) Ad hoc transactions appear at critical APIs in every studied application. Specifically, 71/91 ad hoc transactions are on critical APIs in the studied web applications. For ex-

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is a minor revision of the paper entitled *Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly*, published in SIGMOD '22, ISBN 978-1-4503-9249-5/22/06, June 12–17, 2022, Philadelphia, PA, USA. DOI: 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 redemption.

```
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 operations are shaded yellow; ad hoc constructs are shaded green.

ample, there are 37 ad hoc transactions across 3 e-commerce applications. 31 ad hoc transactions are in critical APIs such as check-out, payment, and add-cart to coordinate operations on critical data (e.g., user credits).

(ii) Ad hoc transactions' usages and implementations are much more flexible than database transactions. For example, 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, upon closer inspection, we found that not all operations require coordination. One reason is that some objects are always associatively accessed, so a single lock is sufficient for ensuring 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. 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. This allows them to 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, achieving up to 1.3× API performance improvement when compared to row-level locking by avoiding false conflicts on the contended rows.

The prevalence of ad hoc transactions and their unique characteristics suggest the potential for 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

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. Most applications manipulate relational data with the help of ORM frameworks such as Hibernate and Active Record. These frameworks can transparently generate SQL statements that fetch and persist data according to the application code. ORMs also provide interfaces to assist developers in coordinating concurrent database accesses: *database transaction APIs* and *invariant validation APIs*.

ORM frameworks usually allow developers to use database transactions explicitly, with interfaces that directly translate to Transaction Start, Commit, and Abort statements. Developers use them to encapsulate multiple database operations into units of work, and the database system takes the responsibility of coordination. A classic application is the coordination of concurrent balance transfer in banking applications. ORM frameworks allow developers to configure the isolation level for specific transactions.

Besides database transactions, ORMs also provide built-in invariant validation APIs. For example, Active Record 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. For example, writing "validates :email, uniqueness: true" tells Active Record to ensure that the email column values are unique. 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.

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 the 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 six categories (Table 1). They are the most popular web applications in each category[1] and written with different languages and different ORMs. For example, Broadleaf is the highest star-ed Java e-commerce application on GitHub

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

Application	Category	Language/ORM	RDBMS	Stars
Discourse	Forum	Ruby/Active Record	PG	33.8k
Mastodon	Social network	Ruby/Active Record	PG	24.6k
Spree	E-commerce	Ruby/Active Record	PG, MY	11.4k
Redmine	Project mgmt.	Ruby/Active Record	PG, MY, +	4.2k
Broadleaf	E-commerce	Java/Hibernate	PG, MY, +	1.5k
SCM Suite	Supply chain	Java/Hibernate	PG, MY	1.5k
JumpServer	Access control	Python/Django	PG, MY, +	16.8k
Saleor	E-commerce	Python/Django	PG, MY, +	13.9k

Table 1: **The applications corpus.** The "RDBMS" column lists supported RDBMSs. "PG/MY/+" refers to PostgreSQL/MySQL/others.

and Spree 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 source code, commit histories, and issue trackers. Then, we manually identify coordination code that isolates database operations and the purpose of those operations.

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

For e-commerce applications, we consider an ad hoc transaction critical if it resides in their core APIs such as checkout 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, Spree, and Saleor, 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. See [13] for descriptions for other applications' core APIs.

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 [5]. 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 [8].

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.*

All DB Operations vs. Specific DB 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.

```
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 The first update changes the quantity in the SKUs table, and other updates refresh the updated_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 [13]. Meanwhile, developers cannot exclude these timestamp updates from the scope of database transactions as the ORM hides the generation of such database operations. Overall, 22 ad hoc transactions coordinate only a portion of the database operations in the transaction scope. Other operations require no coordination but are located in the transaction scope as they are either automatically generated by the ORM or needed by critical operations.

Individual Requests vs. Multiple Requests. It is a performance anti-pattern for database transactions to span multiple HTTP requests, which introduces 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 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* [6] and *savepoints*. Unfortunately, they usually provide (potentially unnecessarily) stronger semantics than what ad hoc transactions provide here [13].

DB Operations vs. Non-DB 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.

```
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 some posts in the RDBMS, which cannot be achieved solely with database transactions. Thus, developers implement ad hoc transactions to coordinate these operations. 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 or XA transactions. However, storage systems rarely support such distributed transaction protocols, which necessitate ad hoc transactions.

3.2 How Is the Coordination Implemented?

Developers need to manually construct ad hoc transactions, which includes manual locking and validation. As a result, 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.*

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, from either existing systems or scratch.

Four applications directly use the locking primitives provided by the database systems or languages runtimes. Specifically, Spree, Saleor, and Redmine use the database Select For Update statements, while SCM Suite 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.

Three other applications, Discourse, Mastodon, and JumpServer, 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 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. As a result, Discourse's Redis lock requires six additional round trips compared to Mastodon's, which only needs one. Saleor uses SETNX to implement locks as Mastodon; it also adds a re-entrant feature, allowing locks to be repeatedly acquired by the same thread.

Broadleaf 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 they are introduced by different developers.

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 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, 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.

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. 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.*

Single Access vs. Multiple Accesses. Locks in ad hoc transactions could coordinate arbitrary database accesses. According to our study, 58 ad hoc transactions use one lock to coordinate multiple database accesses, as developers could manually identify the following two access patterns.

The first pattern is the *associated access* pattern. 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, 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.

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.

The second pattern is the *read–modify–write (RMW)* pattern. RMW means that a transaction first queries the data from the database system, then makes modifications accordingly, and finally persists modifications 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. Consider the example shown in Figure 1b, in the forum application Discourse, 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 this access pattern. Among them, 35 cases also utilize the associated access pattern.

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

There are 5 ad hoc transactions that use column-level coordination. Since 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, 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 their locks do not interfere with each other.

Meanwhile, there are 10 cases that coordinate using predicate information. 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, including MySQL and PostgreSQL. For example, in the Spree 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. We consider these locks a variant of *predicate locks* [5, 7], 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.

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 (19/26 cases) directly return an error to the user on failed validation.*

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.

Rollback methods in ad hoc transactions are either based on (i) database transactions' atomicity property or (2) handcrafted 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, 3]. For example, in Discourse, multiple posts can reference the same image and thus image changes must be applied to all relevant posts. In the course of a background image shrinking job, if a referencing post is updated by the user, instead of aborting the whole process, Discourse uses per-post versions to identify the changed post, only redoes updates for it, and commits the image shrinking process.

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: except in one Broadleaf case, lock information does not persist—they either vanish along with crashes (in-memory locks) or expire after a given period (Redis locks). In Broadleaf, locks are persisted in a database table, accompanied with universally unique identifier (UUID) that distinguishes each application start-up. Thus, Broadleaf can ignore prior unreleased locks after reboot by examining the saved UUIDs.

Meanwhile, the fact that many cases skip rollback (§3.4) indicates that applications might be designed to tolerate intermediate states to a certain extent. For example, every twelve hours, Discourse 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.

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. We have manually verified that all issues are reproducible and cause user-noticeable consequences.

In summary, 69 correctness issues are found in 53 cases; some cases have multiple issues. Furthermore, 28 cases have severe consequences (Table 2), 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[2]) 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 fail to provide atomic validation and commit, causing correctness issues.*

Incorrect Lock Usage. When developers reuse existing systems' locking primitives, misuses arise. Both two reused

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

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

Table 2: **Consequences of incorrect ad hoc transactions.**

existing locking primitives, database systems' Select For Update statements and Java's synchronized keyword (§3.2), have corresponding cases of incorrect usage. For example, in some Spree cases, Select For Update statements are not explicitly enclosed inside database transactions, which causes the database lock to release as soon as the statement returns.

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.

Incorrect Lock Implementation. The locking primitives implemented by developers can also have correctness issues, especially those using Redis or in-memory lock tables. For example, the Redis lock in Mastodon implement the lease semantics, where 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.

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 only when developers manually implement validation procedures (16 cases); cases using ORM-generated validation procedures we studied are all correct.

4.2 Incorrect Coordination Scope

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), it comes with an increased chance of leaving critical operations uncoordinated. For example, in Broadleaf, 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.

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 (e.g., a request handler) without proper ad hoc transactions installed can freely interleave with other ad hoc transaction–coordinated procedures, reading and writing "coordinated" data. For example, in Spree, all ad hoc transactions are deployed in the request handlers that return HTML responses. However, another uncoordinated set of handlers with the same functionality exists and produces JSON responses. As a result, JSON handlers' interleaving with HTML handlers leaves RDBMS states inconsistent.

4.3 Incorrect Failure Handling

FINDING 8. *A minority of issues come from complex coordination, all related to customized failure handling.*

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, when updating image references of posts, developers use versions to detect concurrent modification to fetched posts that use a given image (shown in §3.4). However, their implementation cannot detect newly added posts referencing this image. These new posts will end up having dangling image references, showing end-users broken links. There is the only one 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, it might fail to provide normal services if crashes occur. For example, in Spree, 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. There are 3 cases with similar issues.

5. PERFORMANCE SUMMARY

This section briefly summarize the performance of different designs and implementations of ad hoc transactions using actual application codebases. We refer readers to [13] for more details. 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. Finally, for rollback performance, transaction repair achieves the lowest latency among other rollback methods.

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

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.		

Table 3: **Coordination hints supported by the top ten ranking RDBMSs.** SQLite (6th), MS Access (7th), and Apache Hive (10th) are skipped due to the lack of support for transactions and/or coordination hints.

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, such as multiple storage backends. For example, database transactions surely fall short when business procedures access multiple storage backends (§3.1). Developers have expressed similar concerns [13]. Another potential reason lies in the performance—we have found that ad hoc transactions could perform better than database transactions under contentious workloads (§5). Likewise, developers have expressed performance concerns, e.g., they want to avoid LLTs [13].

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. However, can they help developers write ad hoc transactions or even replace them? To answer this question, we surveyed the supported coordination hints among the top ten ranking RDBMSs[3] and found that they can in part prevent errors while retaining the benefits (Table 3). For example, to coordinate only specific database operations (§3.1), we can augment them with the HOLDLOCK explicit locking hints from SQL Server 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 due to fewer ad hoc constructs.

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. For example, in MySQL, if any table is explicitly locked, accesses to non-explicitly-locked tables are denied; 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. They should include (i) primitives for optimistic ad hoc transactions, (ii) proxy module for existing database coordination hints, and (iii) development support tools. We refer interested readers to [13] for a more detailed discussion.

	Feral CC [1]	**ACIDRain** [15]	**This work**
Target	ORMs' invariant validation APIs	DB transactions	Ad hoc transactions
Aspects	Characteristics Correctness	Correctness	Characteristics Correctness Performance
Issue types	Insufficient isolation	Insufficient isolation Wrong trans. scope	Wrong sync. primitives Wrong trans. scope Wrong failure handling

Table 4: **Comparison with Feral CC and ACIDRain.**

7. RELATED STUDIES

Researchers have studied how database-backed web applications handle concurrency. The major difference between these works and ours lies in the coordination approach being studied. Consequently, we examine different aspects and have arrived at new and interesting findings.

Bailis et al. [1] studied how Rails applications adopt invariant validation APIs to handle concurrency. They have found that application-level invariant validations are used much more often than database transactions. Furthermore, using invariant confluence [2], 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.

Warszawski and Bailis [15] 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 manual inspection, they have identified 22 bugs caused by *insufficient isolation levels* and *incorrect transaction scopes*.

Meanwhile, Xiong et al. [17] surveyed another type of manual coordination—ad hoc loops over synchronization variables in multi-threaded C/C++ 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.

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. 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.

9. 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 SIGMOD '22 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. Zhaoguo Wang (zhaoguowang@sjtu.edu.cn) is the corresponding author.

[3] https://db-engines.com/en/ranking

10. REFERENCES

[1] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *SIGMOD '15*.

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

[3] M. Dashti, S. Basil John, A. Shaikhha, and C. Koch. Transaction repair for multi-version concurrency control. In *SIGMOD '17*.

[4] Z. Dong, C. Tang, J. Wang, Z. Wang, H. Chen, and B. Zang. Optimistic transaction processing in deterministic database. *Journal of Computer Science and Technology*, 35(2):382–394, Mar. 2020.

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

[6] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, Dec. 1987.

[7] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision locks. In *SIGMOD '81*.

[8] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[9] C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX ATC '14*.

[10] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI '12*.

[11] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD '15*.

[12] C. Pu, G. E. Kaiser, and N. C. Hutchinson. Split-transactions for open-ended activities. In *VLDB '88*.

[13] C. Tang, Z. Wang, X. Zhang, Q. Yu, B. Zang, H. Guan, and H. Chen. Ad hoc transactions in web applications: The good, the bad, and the ugly. In *SIGMOD '22*.

[14] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP '13*.

[15] T. Warszawski and P. Bailis. ACIDRain: Concurrency-related attacks on database-backed web applications. In *SIGMOD '17*.

[16] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *SIGMOD '16*.

[17] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI '10*.