DOI:10.1145/3708552

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3708914

Many Faces of Ad Hoc Transactions

By Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen

ABSTRACT

Transactions are the fundamental database abstraction for ensuring application correctness under concurrency and failures. Using database transactions, developers are only tasked with demarcating critical business logic into transactions, and the underlying database system is responsible for coordinating their execution. Yet, in today's Web applications, transactions are often constructed in an ad hoc manner. That is, developers might explicitly use locking primitives or manual validation to coordinate critical code fragments, instead of relying on database transactions. We refer to such application-coordinated transactions as *ad hoc transactions*.

This paper presents the first comprehensive study on ad hoc transactions. Examining 91 ad hoc transaction instances taken from eight popular open source Web applications, we find that (i) every application studied uses ad hoc transactions (up to 16 per application), 71 of which play critical roles; (ii) compared with database transactions, coordination strategies 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, this paper concludes with a discussion about the implications of ad hoc transactions and opportunities for future research.

1. INTRODUCTION

Web applications typically use database systems to store and serve large amounts of data. Therefore, correctly coordinating concurrent database operations is essential for application correctness. The conventional approach is to use database transactions,[8] where developers encapsulate multiple database operations into individual units of work using `Transaction Start/Commit` commands. The database system carefully coordinates their concurrent execution so they appear to run in isolation. In this way, developers are relieved from reasoning about complex concurrent interleavings. Another approach, as identified in Bailis et al.,[2] is to use object-relational mapping (ORM) invariant validation APIs. Developers explicitly specify application invariants, such as column uniqueness, through these APIs, and ORM frameworks check and report errors on invariant violations.

This paper was originally published in *Proceedings of the Intern. Conf. on Management of Data*, ACM (June 2022).

Besides these two approaches, we have observed that application developers often manually coordinate critical database operations. They might explicitly use locking primitives or validation procedures to implement concurrency control (CC) amid the application code. We refer to such ad hoc coordination of database operations as *ad hoc transactions*.

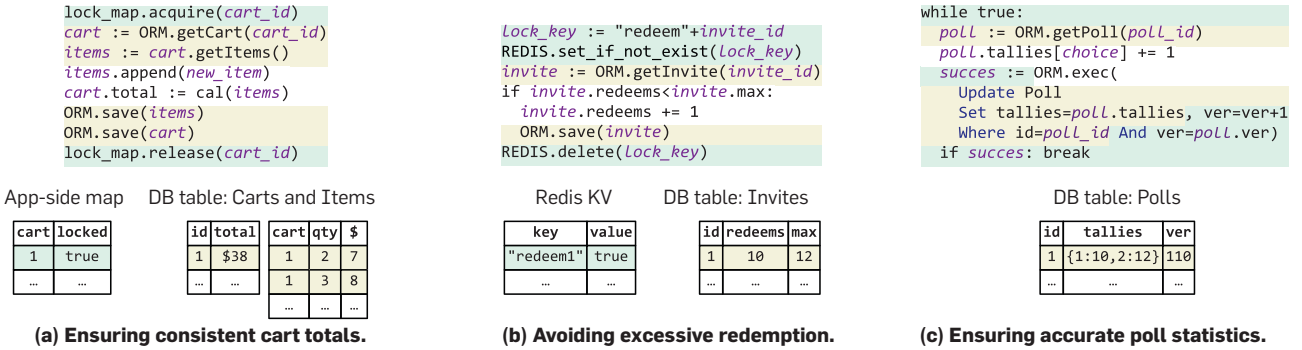
Figure 1 shows three real-world examples of ad hoc transactions from open source Web applications. In each example, the application code issues database operations via ORM frameworks and uses ad hoc constructs to coordinate them. The first two directly use locks for coordination, while the third one implements a validation-based protocol similar to optimistic concurrency control (OCC). We briefly explain Figure 1a, an `add-to-cart` code snippet from an e-commerce application. First, a lock identified by the cart ID is acquired and held until the whole business logic finishes. Then, database rows for the cart and corresponding items are retrieved and converted into runtime objects `cart` and `items` by the ORM. Next, a new item is added to the `items` collection, and the total price of the `cart` object is recalculated. Finally, the ORM persists two updated objects back to the database system.

As shown in these examples, ad hoc transactions are tightly 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 have been clearly understood. While some developers' comments might suggest they implement ad hoc transactions for flexibility or efficiency reasons, it is unclear whether ad hoc transactions meet these expectations and at what cost.

To address this gap, we spent five person-years conducting a comprehensive study over 91 ad hoc transactions among eight Web applications from 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). At a high level, we discovered the following interesting, perceptive, and potentially alarming findings.

Every application studied uses ad hoc transactions on critical APIs: Specifically, 71 out of 91 ad hoc transactions are on the critical APIs in the Web applications studied. For example, there are 37 ad hoc transactions across three e-commerce applications. Thirty-one ad hoc transactions are in critical APIs, such as `check-out`, `payment`, and `add-cart` to coordinate operations on critical data (for example, user credits).

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



(a) Ensuring consistent cart totals. **(b) Avoiding excessive redemption.** **(c) Ensuring accurate poll statistics.**

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

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. In addition, both application server crashes and database system crashes can easily cause inconsistent states and ultimately buggy behavior. Among all issues, incorrect synchronization primitive implementations are the most common cause (47 cases). We have submitted 20 issue reports (covering 46 cases) to developer communities; seven issues (covering 33 cases) have been acknowledged.

Ad hoc transactions can have performance benefits under high-contention workloads: Using application semantics such as access patterns, ad hoc transactions' concurrency control (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 suggests the potential for improving existing database systems that support these applications. We conclude by discussing the implications of our findings on future database- and storage-systems research. This paper condenses our results from the previous conference version[11] and the extended journal version,[12] and we refer interested readers to the full versions for more details.

2. BACKGROUND AND MOTIVATION

Concurrency control in Web applications.

Most Web applications manipulate 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 based on how applications manipulate in-memory objects. 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. Coordinating concurrent money transfers that consist of reading and writing common account balances is a textbook example of using database transactions. Meanwhile, ORMs also provide built-in invariant validation APIs. For example, with the Active Record library developers can specify column uniqueness as "`validates:col , uniqueness: true`". Before persisting an object at runtime, the library checks if there is an existing database row with the same `col` value, and reports errors on duplication.

Ad hoc transactions in the wild.

Ad hoc transactions are the third coordination approach. Like database transactions, ad hoc transactions provide isolation to concurrent database operations. However, their coordination is manually programmed by application developers instead of relying on database systems. Like ORM's invariant validation APIs, ad hoc transactions operate at the application level. However, the former works by indirectly checking database states for violations while the latter directly coordinates conflicting database operations.

To understand ad hoc transactions' roles and importance in Web applications, we investigated eight representative applications of six categories (Table 1). They are the most popular Web applications in each category[a] and are written in different languages and different ORMs. We locate ad hoc transactions by first searching keywords, such as "lock," "concurrency," and "consistency" in source code, commit histories, and issue trackers for potential instances. Then, we

a Redmine now is the second most popular project management application. Its popularity has waned since we chose it as the investigation target.

manually examine the coordination code that isolates database operations and the purpose of those operations.

Table 1. The applications corpus. The "RDBMS" column lists supported relational database management systems (RDBMSs). "PG/MY/+" refers to PostgreSQL/MySQL/others.

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

Finding 1.
Every application studied uses ad hoc transactions. There are 91 ad hoc transactions identified in total and 71 cases are considered critical to the Web applications.

Many ad hoc transactions are used in important APIs. In e-commerce applications, such APIs include `check-out` and `add-cart` where ad hoc transactions ensure business safety, for example, by coordinating 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 five avoid inconsistent capture of payment. Interestingly, all these applications have ad hoc transactions to ensure sufficient stock quantity and coupon validity. For important APIs in other categories, see Tang et al.[11] and Wang et al.[12]

3. CHARACTERISTICS OF AD HOC TRANSACTIONS

We have carefully studied the 91 identified ad hoc transaction cases. An interesting but less 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.[6] Unlike database transactions, pessimistic ad hoc transactions' locking primitives are usually implemented from scratch by application developers (for example, 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.[10] We dissect ad hoc transactions in the following subsections regarding what they protect, how they protect, and how they handle failures. A detailed comparison with database transactions is available in Wang et al.[12]

3.1. What do ad hoc transactions coordinate?

In writing ad hoc transactions, developers explicitly place ad hoc coordination constructs among the business logic. Developers are thus flexible in choosing which and how operations are coordinated, resulting in interesting patterns such as partial coordination, cross–HTTP request coordination, and coordination with non-database operations.

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

All DB operations vs. specific DB operations. With ad hoc transactions, 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 following 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, and 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 (lines 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 `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. Interestingly, replacing the `lock()/unlock()` primitives with `Transaction Start/Commit` may worsen performance due to indiscriminately upgrading all operations to the same isolation level. Furthermore, 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. Database transactions spanning multiple HTTP requests are a performance anti-pattern, as they introduce long-lived transactions (LLTs). Yet, we have found 10 ad hoc transactions with life spans 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.

```
1 Request 1 // fetch a post & increment view count
2  in: post_id
3  Update Post Set view_cnt=view_cnt+1,
   ver=ver+1 Where id=post_id
4  post := Select * From Posts Where
   id=post_id
5  response render(post) // this response
   includes the version
6 Request 2 // detect interruptions & apply user
  updates
7  in: post_id, new_content , prev_ver
8  lock(post_id)
9  current := Select * From Posts Where
   id=post_id
10 if current.ver!=prev_ver: unlock(post_id);
   response FAILURE
11 Update Posts Set content=new_content ,
   ver=ver+1 Where id=post_id
12 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 number with each post to track updates. Before updating a post, the ad hoc transaction checks the consistency (that is, 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. Interestingly, unlike database transactions that undo all effects if aborted, this ad hoc transaction does not roll back the view count increment made in the previous request handler. 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*[7] and *savepoints*. Unfortunately, they usually provide (potentially unnecessarily) stronger semantics than what ad hoc transactions provide.

DB operations vs. non-DB operations. Ad hoc transactions' flexibility also shines in coordinating non-database operations. A Web application may use multiple storage systems. Thus, it needs to ensure data consistency across different systems. There are eight cases of ad hoc transactions that coordinate both database operations and non-database operations, such as operations over in-memory shared variables, local file systems, and remote object/key–value (KV) stores. Consider the following example of the timeline feature from the Mastodon social network application.

```
1 Create Post
2  in: follower_id, post_id, content
3  lock(post_id)
4  Insert Into Posts Value (post_id, content)
5  REDIS.add_to_set("timeline"+follower_id,
   post_id)
6  unlock(post_id)
7 Delete Post
8  in: follower_id, post_id
9  lock(post_id)
10 REDIS.delete_from_
   set("timeline"+follower_id, post_id)
11 Delete From Posts Where id=post_id
12 unlock(post_id)
```

It uses a Redis KV store and an RDBMS as its back-end storage. Redis holds the IDs of posts shown on each user's timeline, while the concrete post contents are resident in the RDBMS. Mastodon must guarantee 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, and this cannot be achieved solely with database transactions. Thus, developers implement ad hoc transactions to coordinate these operations. In general, when applications require data consistency across multiple storage systems (including multiple RDBMSs), the alternative option is to use distributed transactions, such as WS-TX or XA transactions. However, storage systems rarely support such distributed transaction protocols, making writing ad hoc transactions the only feasible solution.

3.2. How is the coordination implemented?

Given the manual nature of ad hoc transactions, it is not surprising that application developers have implemented many flavors of coordination primitives.

FINDING 3.
There are seven different lock implementations and two validation implementations among the eight applications we studied. Except for Broadleaf, developers consistently use the same lock/validation implementation in individual applications.

Locks from existing systems vs. manually crafted locks. All eight applications studied 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 language 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 Jump-Server—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 roundtrips 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 from-scratch lock implementations and existing systems' primitives—the Java `synchronized` keyword. More interestingly, it has three from-scratch implementations: One uses a separate database table to store lock information similar to 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. These implementations are introduced by different developers, and we have not found clear evidence that they serve different purposes.

ORM-assisted validation vs. manual validation. Six out of eight applications studied have validation-based optimistic ad hoc transactions. Their validation procedures are either provided by the ORM or the developers themselves.

Four applications 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?

With ad hoc transactions, developers can customize the coordination granularity. Intuitively, one might think of using *finer-grained coordination* rather than database transactions to improve parallelism. However, we have found that ad hoc transactions also employ *coarser-grained coordination*, by grouping multiple operations and coordinating them with a single lock. This can largely reduce ad hoc transactions' CC complexity and avoid deadlocks.

> Finding 4.
> Among the 91 ad hoc transactions identified, 14 cases perform fine-grained coordination, such as column-based coordination, while 58 cases perform coarse-grained operations—that is, using a single lock to coordinate multiple operations. Nine cases implement both types of coordination for different accesses.

Single access vs. multiple accesses. Fifty-eight ad hoc transactions use only one lock to coordinate multiple database accesses, thanks to the following two access patterns in the applications.

Associated access: 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. 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. This pattern provides an opportunity to replace multiple locks (for example, row locks) with one lock that coordinates these accesses. In the above example, developers use a single cart lock to coordinate accesses to both `Carts` and `Items` tables. 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. In all these cases, the associated rows are connected by either one-to-many or one-to-one relationships. We find that such one-to-many relationships all stem from the application-specific data modeling that reflects the business semantics, like the cart-item relationship in the example. Meanwhile, such one-to-one relationships all come from inheritance.

Read–modify–write (RMW): With RMW, a transaction first reads data from the database, then modifies it, and finally writes it back. In a typical 2PL system such as MySQL, deadlocks arise if concurrent transactions perform 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 concur-

rently use the same 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 read, avoiding possible deadlocks. Fifty-six out of 91 cases leverage this access pattern. Among them, 35 cases also use the associated access pattern.

Fine-grained vs. coarse-grained. Ad hoc transactions can also use fine-grained column- or predicate-based coordination.

Knowing which columns are used in the business logic allows developers to coordinate database operations at the column granularity, and there are five such ad hoc transactions. 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 their locks do not interfere with each other.

Likewise, knowing the search conditions, developers can use the precise predicate for coordination, and there are 10 such cases. Predicate-based coordination 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 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. By manually locking on specific `order_id` values, developers can avoid such false conflicts.

3.4. How are failures handled?

Similar to database transactions, ad hoc transactions are faced with both runtime failures and system crashes.

> FINDING 5.
> Ad hoc transactions are typically not equipped with complex failure-handling logic, partly because individual ad hoc transactions might face fewer potential failure scenarios than general database transactions and partly because developers seem to often assume failure-free executions.

Automated rollback vs. manual rollback. We first consider runtime failures, which include deadlocks or validation failures. Unlike database transactions, which have a system-level catch-all rollback mechanism, application developers need to craft failure-handling logic on a case-by-case basis.

Each pessimistic ad hoc transaction either uses a single lock (52/65) or acquires locks in a consistent order (13/65). Thus, none needs to handle deadlock at runtime. The same applies to locking in optimistic ad hoc transactions. Interestingly, some optimistic cases do not acquire any lock during the validate-and-commit process, which eliminates deadlocks but sacrifices correctness (see §4.1).

Therefore, rollback is only needed for validation failures in optimistic ad hoc transactions. In 19 optimistic cases, developers follow the classic OCC paradigm, where no update is persisted before validation, thus avoiding the need for rollback. In other cases, early persisted updates require developers to either use certain *rollback methods* to negate the effect of updates or use *repair techniques* to roll forward and finalize changes.

To roll back, ad hoc transactions rely on either database transactions' atomicity property or hand-crafted rollback procedures. There is one case using the former approach, where a `Read Committed` database transaction is used to enclose update and validation operations. If validation fails, the application aborts the database transaction to roll back all enclosed updates. Meanwhile, two cases use manually programmed rollback procedures triggered by validation failures and will undo persisted updates. Unlike database rollback, developers can selectively undo only critical updates. For example, in Broadleaf, if validation on SKU state fails during checkout, updates to payment and order status

are rolled back, while other updates like total order price calculation are kept unaffected.

Meanwhile, four 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 recently proposed transaction repair optimizations.[4,5,14] For example, in Discourse, multiple posts can reference the same image and thus changes to images must be propagated to all relevant posts. During 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. Since ad hoc transactions are not automatically rolled back upon crashes, developers need to manually ensure coordination metadata (for example, locks) and database states are properly recovered. Among the former case, version numbers and most lock states require no special handling, as optimistic ad hoc transactions can always retry reading the latest version, and both in-memory and Redis-based locks do not (forever) persist across crashes. However, the database-based persistent locks in Broadleaf require special handling. To avoid deadlock caused by unreleased locks, developers associate each lock using a boot-time generated universally unique identifier (UUID) to distinguish each run. Thus, Broadleaf can ignore prior unreleased locks after reboot by examining the saved UUIDs.

Recovering database states after crashes is more challenging than validation failure rollback since developers are unaware of the progress of interrupted ad hoc transactions. Though in the case of database crashes, applications can spin-wait for database restart and then either roll back or continue, all ad hoc transactions simply terminate immediately and leave the database in an intermediate state. Only one application, Discourse, has a `fsck`-like checker that runs every 12 hours to actively restore inconsistent references, such as missing avatars and thumbnails. Other applications either tolerate intermediate states with preventative measures or simply let the inconsistency propagate to end users (§4.3). An example of preventative measures is Broadleaf, which sets any payment record left unconfirmed during an interrupted checkout before starting a new one.

4. CORRECTNESS ISSUES

Building bug-free ad hoc transactions is nontrivial. In this section, we summarize the correctness issues we have found and relate them to the ad hoc transaction 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 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) to developer communities; seven of them (covering 33 cases) have been acknowledged.

Table 2. Consequences of incorrect ad hoc transactions.

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

4.1. Incorrect locks and validation procedures.

FINDING 6.
Thirty-six 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. Issues exist for both types of locking primitive reuse: `Select For Update` from database systems and Java's `synchronized` keyword (§3.2). 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 use a single lock to coordinate RMW operations, and they omit the first query from protection. This happens often when developers have to first perform the query to obtain the lock key, for example, an ID. In these situations, developers should re-execute the query after acquiring the lock to protect the entire RMW. There are two cases where the developers forget to re-execute queries, leaving the initial read in RMW uncoordinated.

Incorrect lock implementation. The locking primitives implemented from scratch can also have correctness issues, especially those using Redis or in-memory lock tables. For example, the Redis lock in Mastodon implements 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); all cases using ORM-generated validation procedures we studied are correct.

4.2. Incorrect coordination scope.

FINDING 7.
Sixteen 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 (for example, 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.
Ad hoc transactions are often vulnerable to incorrect handling of both runtime failures and server crashes.

Incomplete repair. When using transaction repair to roll forward an interrupted 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 only one case that has this issue.

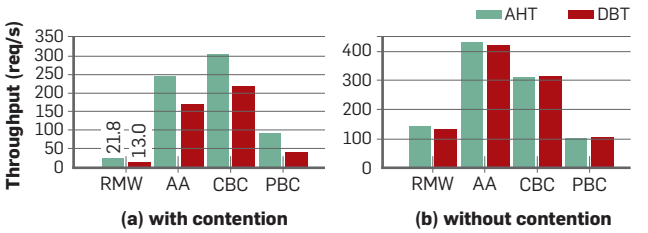
Unexpected intermediate states after crashes. If an application is not designed to tolerate intermediate database states, and rollback handlers fail to prevent such states, the application might fail to provide normal services if server or database crashes occur. We thoroughly investigated the impact of crashes in ad hoc transactions from Broadleaf and Spree. We identified 31 unique crash scenarios where a crash leaves writes partially executed and found that 28 of these scenarios have user-noticeable unexpected behaviors. For example, in Spree, a crash during check-out can leave payments in an intermediate state (that is, 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 under "processing" by active threads. Therefore, users can never finish the check-out.

5. PERFORMANCE IN ACTION

We have evaluated the performance of different designs and implementations of ad hoc transactions using actual application codebases, and the results are summarized below. First, there are order-of-magnitude performance differences between different primitive implementations. Disk I/Os and network roundtrips are the key 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, among all failure-handling approaches, repair achieves the lowest latency.

Due to space constraints, we only present the results with different coordination granularities (§3.3). We evaluated four real-world APIs, where the four customized 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 code base that uses ad hoc transactions (denoted as `AHT`) and a modified one that instead uses database transactions under the weakest yet sufficient isolation level (denoted as `DBT`). As shown in Figure 2, under contentious workloads, `AHT` achieves up to 1.3× higher throughput than `DBT`, and the geometric mean of improvements is 63.0%. Under no-contention workloads, `AHT` and `DBT` have similar performance. These results confirm our hypothesis on the potential benefits of using customized coordination granularities. Specifically, in `RMW` and `AA`, acquiring locks early and aggressively prevents deadlocks in MySQL and write–write conflicts in PostgreSQL. As a result, conflicting API requests' non-critical sections are effectively pipelined with the one active critical section, improving CPU efficiency. Meanwhile, by coordinating at a more fine-grained and precise level, `CBC` and `PBC` avoid false conflicts that would arise in database transactions.

Figure 2. API throughputs using different coordination granularities.



6. DISCUSSION

We have shown that ad hoc transactions are error-prone and difficult to identify and understand, but they are still

widely used in Web applications, mostly among critical APIs. Besides this study, others have also reported the use of application-level manual coordination in large-scale Web applications, including Taobao,[b] the largest e-commerce platform in China.[2,3,9] Rather than attributing such practice to developers collectively overlooking the power of database transactions, we argue there is an emerging gap between the coordination requirements of Web applications today and what database systems currently offer. Functionality-wise, as shown in §3.1 and 3.4, certain business logic using ad hoc transactions exhibits characteristics that are difficult, sometimes impossible, for database transactions to handle, such as the use of multiple storage backends and life span crossing multiple requests. Performance-wise, as shown in §5, the flexibility of ad hoc transactions allows developers to tailor coordination for individual APIs, which can sometimes outperform general, one-size-fits-all database transactions.

To address this gap, we envision a three-step approach. First, we need to systematically quantify the gap. This paper focuses mainly on the characteristics of ad hoc transactions themselves and provides only an initial answer to the "what-if" question: what if developers choose to work around database transactions instead? We believe further study on both development efforts and runtime overheads is still required. Only with concrete measurements can we identify promising future directions.

Next, we need to explore incremental remedies for existing applications. While it is tempting to propose new abstractions or tools from scratch, incremental remedies are often a better fit for existing running applications. This consideration is partly motivated by the fact that many existing database systems already provide interfaces for passing hints that customize the coordination. As summarized in Table 3 and in Tang[11] and Wang,[12] we have surveyed such coordination hints among the top ten RDBMSs in the db-engines.com ranking and found that they can potentially prevent some of the ad hoc transaction errors while retaining their benefits. Therefore, we believe it is promising to explore new ways to enhance existing applications using ad hoc transactions with these tools. One possible direction would be to propose proxy modules that hide the differences among different database systems and provide general coordination customization interfaces.

Finally, we should explore alternative abstractions for emerging applications. With the increasing complexity of modern Web applications, new system architectures (such as microservices and serverless) and new computation patterns (such as cloud-edge collaboration and artificial intelligence of things), are becoming the new norm. In this complicated context, concurrency control is ever more important. This work and others[9] have observed that such applications are departing from the classic ACID transaction semantics. Therefore, instead of focusing on intra-component coordination driven by individual database systems, new global abstractions that address inter-component concurrency are urgently needed. For example, how to formalize non-ACID semantics in a way that is intuitive and useful to developers while allowing efficient system implementations is still an open question.

7. RELATED STUDIES

Researchers have studied various other approaches, such as database-backed Web applications use to handle concurrency. Bailis et al.[2] 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,[1] they have found that most of the validations are sound, that is, they preserve invariants even under concurrent execution using weak isolation levels, such as `Read Committed`, while the remainders do not. Warszawski and Bailis[13] focused on whether database transactions are correctly used 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. Cheng et al.[3] examined concurrency-related bug reports of open-source Web applications to understand their root causes, consequences, and fixes. Consistent with our findings, they have found that developers typically opt for manual, ad hoc solutions instead of resorting to strongly isolated database transactions.

8. CONCLUSION

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

b How do cloud users use cloud database systems? (in Chinese) `https://tinyurl.com/2aefkw3u`

Table 3. Coordination hints supported by the top 10 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.

Coordination Hints	Oracle	MySQL, MariaDB	SQL Server, Azure SQL	PostgreSQL	IBM Db2
Explicit table locks	✓ They have different restrictions (for example, syntax) and behaviors (for example, 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.		

popular open-source Web applications, highlighting both the prevalence and significance of ad hoc transactions. While ad hoc transactions offer greater flexibility compared to traditional database transactions, this flexibility is a double-edged sword—providing potential performance benefits but also increasing the risk of correctness issues.

9. ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61925206, 62132014, and 62272304.

References

1. Bailis, P. et al. Coordination avoidance in database systems. In *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 185–196.
2. Bailis, P. et al. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD Intern. Conf. on Management of Data*. Association for Computing Machinery, 1327–1342.
3. Cheng, C. et al. Developer's responsibility or database's responsibility? Rethinking concurrency control in databases. In *Proceedings of the 13th Biennial Conf. on Innovative Data Systems Research* (2023).
4. Dashti, M., Basil John, S., Shaikhha, A., and Koch, C. Transaction repair for multi-version concurrency control. In *Proceedings of the 2017 ACM Intern. Conf. on Management of Data*. Association for Computing Machinery, 235–250.
5. Dong, Z. et al. Fine-grained re-execution for efficient batched commit of distributed transactions. In *Proc. VLDB Endow. 16*, 8 (Apr. 2023), 1930–1943.
6. Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
7. Garcia-Molina, H. and Salem, K. Sagas. *SIGMOD Rec. 16*, 3 (Dec. 1987), 249–259.
8. Gray, J. The transaction concept: Virtues and limitations. In *Proceedings of the 7th Intern. Conf. on Very Large Data Bases 7*, VLDB Endowment, (1981), 144–154.
9. Helland, P. Life beyond distributed transactions: An apostate's opinion. *ACM Queue 14*, 5 (oct 2016), 69–98.
10. Kung, H. T. and Robinson, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.
11. Tang, C. et al. Ad hoc transactions in web applications: The good, the bad, and the ugly. In *Proceedings of the 2022 Intern. Conf. on Management of Data*, Association for Computing Machinery, (2022), 4–18.
12. Wang, Z. et al. Ad hoc transactions through the looking glass: An empirical study of application-level transactions in web applications. *ACM Trans. Database Syst. 49*, 1, feb 2024.
13. Warszawski, T. and Bailis, P. ACIDRain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM Intern. Conf. on Management of Data*, Association for Computing Machinery (2017), 5–20.
14. Wu, Y., Chan, C.-Y. and Tan, K.-L. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 Intern. Conf. on Management of Data, SIGMOD '16*. Association for Computing Machinery (2016), 1689–1704.

Chuzhe Tang is a Ph.D. candidate at the Institute of Parallel and Distributed Systems at Shanghai Jiao Tong University, Shanghai, China, and a member of the Engineering Research Center for Domain-Specific Operating Systems (Ministry of Education), Shanghai, China.

Zhaoguo Wang is an associate professor at the Institute of Parallel and Distributed Systems at Shanghai Jiao Tong University, Shanghai, China, and a member of the Engineering Research Center for Domain-Specific Operating Systems (Ministry of Education), Shanghai, China.

Xiaodong Zhang was a master's student at the Institute of Parallel and Distributed Systems at Shanghai Jiao Tong University, Shanghai, China, and a member of the Engineering Research Center for Domain-Specific Operating Systems (Ministry of Education), Shanghai, China, at the time of writing.

Qianmian Yu was a master's student at the Institute of Parallel and Distributed Systems at Shanghai Jiao Tong University, Shanghai, China, and a member of the Engineering Research Center for Domain-Specific Operating Systems (Ministry of Education), Shanghai, China, at the time of writing.

Binyu Zang is a professor at the Institute of Parallel and Distributed Systems at Shanghai Jiao Tong University, Shanghai, China, and a member of the Engineering Research Center for Domain-Specific Operating Systems (Ministry of Education), Shanghai, China.

Haibing Guan is a professor at Shanghai Jiao Tong University, Shanghai, China, and the director of the Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai, China.

Haibo Chen is a distinguished professor and director of the Institute of Parallel and Distributed Systems at Shanghai Jiao Tong University, Shanghai, China, and the director of the Key Laboratory of System Software (Chinese Academy of Sciences), Beijing, China.