Are Database System Researchers Making Correct Assumptions About Transaction Workloads?

CUONG D. T. NGUYEN, University of Maryland, USA
KEVIN NIECHEN, University of Maryland, USA
CHRIS DECAROLIS, University of Maryland, USA
DANIEL J. ABADI, University of Maryland, USA

Many recent papers have contributed novel concurrency control and transaction processing algorithms that start by making an assumption about the transaction workload submitted by an application, and yield high performance (sometimes by an order of magnitude) under these assumptions. Two of the most common assumptions are (1) Is the read and write set of a transaction known (or easily derivable) directly by analyzing the application code in advance of transaction execution, or is the access set of a transaction dependent on the current state of the database? (2) Does the application send the entire transaction in a single request to the database system or is the transaction sent via several requests "interactively", with application code run in between these requests. The database community has made tremendous progress in improving throughput and latency of transaction processing when read/write sets are known in advance, and for non-interactive transactions. However, the impact of this progress is directly dependent on the accuracy of these assumptions both for current and future applications. In this paper, we conduct an extensive study of 111 open-source applications, analyzing over 30,000 transactions to evaluate the accuracy of these assumptions both as they exist in the current codebase, and how extensive are the changes required to the code for these assumptions to hold moving forward. Our study reveals that the second of these assumptions is stronger than the first. More specifically, for 90% of applications, at least 58% of transactions per application have read/write sets that can be inferred in advance. Furthermore, although only 39% of applications contain zero interactive transactions, nonetheless, the majority of the remaining 61% of applications can be converted to being completely non-interactive with minimal changes. These insights underscore the potential for further optimization and research in designing OLTP systems that balance transaction expressivity and performance.

CCS Concepts: • **Information systems** → **Database transaction processing**; *Relational database model*; • **General and reference** → **Surveys and overviews**.

Additional Key Words and Phrases: transaction processing, workload assumptions, transaction interactivity, read/write set inference, object-relational mapping

ACM Reference Format:
Cuong D. T. Nguyen, Kevin Niechen, Chris DeCarolis, and Daniel J. Abadi. 2025. Are Database System Researchers Making Correct Assumptions About Transaction Workloads?. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 131 (June 2025), 26 pages. https://doi.org/10.1145/3725268

1 Introduction

Inspired by the pioneering work of System R [4] and Ingres [40] in the 1970s, many database systems present an interactive interface that allows a client to issue multiple separate commands to the database system that are intended to be executed atomically and included within a single

Authors' Contact Information: Cuong D. T. Nguyen, University of Maryland, College Park, USA, ctring@umd.edu; Kevin Niechen, University of Maryland, College Park, USA, kev@umd.edu; Chris DeCarolis, University of Maryland, College Park, USA, cdguitar817@gmail.com; Daniel J. Abadi, University of Maryland, College Park, USA, abadi@umd.edu.

ACM 2836-6573/2025/6-ART131
https://doi.org/10.1145/3725268

transaction; potentially running arbitrary application code locally in between the issuance of these commands. Typically, the client initiates a transaction by sending a `BEGIN` command to the server, followed by multiple data access commands. The transaction concludes with either a `COMMIT` or `ABORT`.

However, such interactive transactions, that involve multiple round trips between the application and the database server limit the performance of transactional processing [41]. Every round trip between the application code and the database server pays the overhead of client-server communication. Furthermore, the isolation guarantees of the database may prevent concurrent transactions that access the same data that has been accessed thus far by an interactive transaction from running. These other concurrent transactions may be delayed for an unknown length, since only the application has control over when the final command for an interactive transaction will be issued, allowing the database to complete the transaction and stalled conflicting transactions to run. Moreover, as database research introduces concurrency control algorithms that plan an entire batch of transactions in advance of their execution [12, 13, 33, 44], the need to have the entire transaction available during the planning stages is becoming increasingly important.

Therefore, a large number of recently designed systems have placed limits on transaction interactivity. Some systems completely eschew interactive transactions, supporting only transactions executed in one round of communication between the client and the server (henceforth referred to as **one-shot transactions**). Examples of such systems include H-Store [22], Granola [8], Calvin [44], Silo [46], BOHM [12], Janus [30], PWV [13], STAR [29], Ocean vista [14], Chiller [52], Tempo [10], DynamoDB [19], and more [3, 6, 9, 17, 20, 21, 25–28, 31, 33–38]. Other systems still support interactive transactions, but are optimized for those with one or a few round-trips, such as Carousel [50] and Natto [51].

Systems that plan execution of batches of transaction in advance typically make a second assumption, beyond non-interactivity. They assume that the complete read/write set of every or most transactions is known prior to scheduling and executing any part of these transactions. We refer to these system as **upfront access-knowledge systems** throughout this paper. This *a priori* knowledge about the read/write set can be used to produce more efficient execution schedules, such as those that are deadlock-free. These read/write sets are either declared explicitly or extracted using static analysis [20]. However, some transactions are too complex for static analysis to work. Furthermore, even simple transactions may not know their access-sets in advance if the determination of which data will be accessed is dependent on the current state of the database [43]. Instead, fallback mechanisms, such as Optimistic Lock Location Prediction (OLLP) [39, 44] must be used for these types of *data-dependent transactions*

The upfront access-knowledge assumption is commonly made in deterministic database systems [2], such as Calvin [44], VLL [38], BOHM [12], PWV [13], SLOG [37], Q-Store [34], Caracal [35], Detock [31], and Caerus [17]. Other systems adopting this assumption include Janus [30], Carousel [50], Ocean Vista [14], Strife [33], Tempo [10], and Natto [51]. All these systems use some form of fallback mechanism when the access set is not known in advance.

Both the non-interactivity assumption and the upfront access-knowledge assumption trade transaction expressivity for performance gains. The more accurate the assumption, the more significant the performance gain. It is thus important to understand the accuracy of these assumptions in modern application workloads.

We therefore conducted a comprehensive study of 111 open-source applications, collectively containing more than 30,000 transactions, to investigate how often these assumptions hold in practice. We focus on highly popular applications, measured by GitHub stars, built with Django (Python) and TypeORM (TypeScript), which are among the most popular Object-Relational Mapping (ORM) frameworks of their respective language. These ORMs were chosen because they expose

database operations as small well-documented sets of methods, enabling automated tools to quickly identify these database operations within large codebases, thereby enabling the large scale of this study. Previous studies on real-world transactions [5, 42] have also targeted ORM-backed applications to gain insight into transaction usage in general.

We aim to answer the following questions.

(1) Is it reasonable to assume that the read/write set of real-world transactions can be known in advance?
(2) How common are data-dependent transactions?
(3) How often are interactive transactions used in existing applications?
(4) How hard would it be to modify existing interactive transactions to remove their interactivity?

To answer the first two questions, we propose four different definitions of the access-set, each with varying trade-offs regarding the granularity of conflict detection among transactions and the ability to extract the read/write set statically from transaction code without resorting to a *fallback mechanism*, as detailed in Section 4.1. We then quantify the degree of fallback for each definition by analyzing code of every transaction and annotating the properties that necessitate fallbacks under each definition. To answer the latter two questions, we identify whether a transaction is interactive or not and further categorize interactive transactions into those that can be converted to one-shot transactions and those that cannot due to the presence of *external operations*, described in Section 5.2.

The results of our study show that for 90% of applications, at least 58% of their transactions have read/write sets that are possible to infer in advance. This means that at most 42% of transactions in these applications require some fallback mechanism to determine the read/write set. Furthermore, the data suggests that these fallback mechanisms can be lightweight, potentially resulting in low overhead. The study also finds that 39% of the applications contain zero interactive transactions in their present state. Among applications that do contain interactive transactions, only an average of 9.6% of their transactions are interactive, and almost all of them can be converted into one-shot transactions with minimal modifications. These results highlight research opportunities to further enhance systems relying on these assumptions.

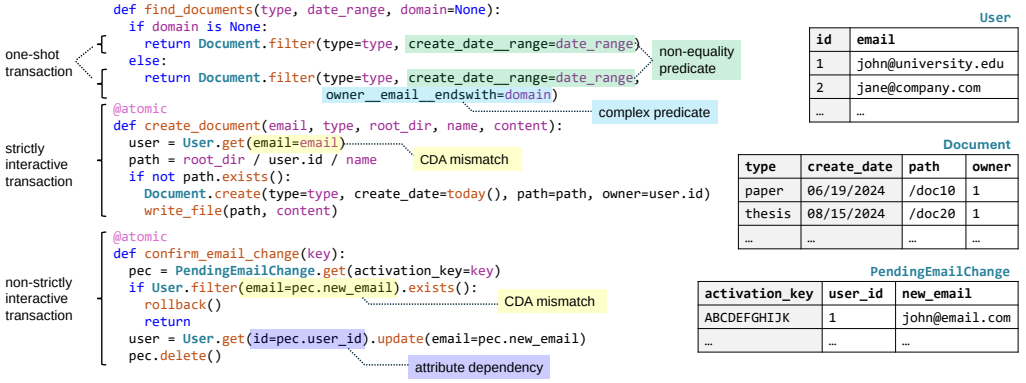

Fig. 1. An example of a document management application using Django (API simplified for brevity).

2 Object-Relational Mapping

Web applications are often written in object-oriented languages, where data are modeled as a collection of objects that interact with each other. In contrast, relational database systems store data

in structured tables, which are accessed and manipulated using SQL. To bridge the gap between these two paradigms, developers can employ an Object-Relational Mapping (ORM) framework.

ORM frameworks translate object-oriented concepts into SQL queries and map the retrieved data back to corresponding objects in memory. This approach enables developers to write both data definition and data manipulation logic in the same language as the application itself. In our study, we focus on applications built with two popular frameworks: Django (Python) and TypeORM (TypeScript), which are widely used within their respective ecosystems.

When using an ORM, developers define special classes called *models*[1], each of which is typically mapped to a database relation. The fields within these classes correspond to the relations' attributes. ORM frameworks also provide mechanisms for defining relationships between fields of different models. Instead of writing raw SQL requests, developers interact with the underlying data by calling methods on objects instantiated from these classes. Some methods manipulate in-memory data on the client side, while others generate and submit SQL statements to the database server. We refer to such submissions as **operations**.

Additionally, ORMs offer APIs to group multiple operations into a single transaction. Such transactions are generally "interactive" since each component operation is sent separately to the database system, thereby resulting in multiple round trips. When operations are not combined using such APIs, a transaction is sent to the database system consisting of a single statement (which we called one-shot transactions in Section 1). Throughout the rest of this paper, we use the term **transaction** to refer to either an interactive transaction or a one-shot transaction.

Fig. 1 illustrates an example document management application that uses Django ORM (some labels, such as "Strictly Interactive" and "CDA Mismatch", will be defined in later sections). This application includes 3 models: `User`, `Document`, and `PendingEmailChange`. The `id` attribute of the `User` model has a one-to-many relationship with both the `owner` attribute of the `Document` model and the `user_id` attribute of the `PendingEmailChange` model.

The `find_document` function contains two separate database operations that filter documents by type, creation date, and optionally, the domain of the owner's email. The `create_document` function, decorated with `@atomic`, executes as an interactive transaction. It creates a new document on disk with the provided metadata and content and adds a corresponding entry to the `Document` model. The `confirm_email_change` function, also an interactive transaction, updates a user's profile with a pending email change after the user clicks a confirmation link containing an activation key.

3 Data Collection

This section outlines the steps taken to choose which open-source applications to analyze, annotate each application with its individual database operations, and aggregate statistics from these annotations to form the core of our dataset, the results of which are presented in Table 2 and 3. To facilitate this process, we developed a tool[2] that automates the annotation for simple cases and provides a user interface for manual verification and handling of more complex scenarios that cannot be reliably automated.

3.1 Application Corpus

To compile a set of open-source applications to analyze, we leveraged the GitHub code search API and identified repositories containing keywords indicative of Django ORM usage (e.g., `django.db`) or TypeORM usage (e.g., `createQueryBuilder`). Using GitHub stars as a measure of open-source

[1]Referred to as "entities" in TypeORM

[2]The tool is written as an extension for Visual Studio Code and can be found at https://github.com/umd-dslam/splinter

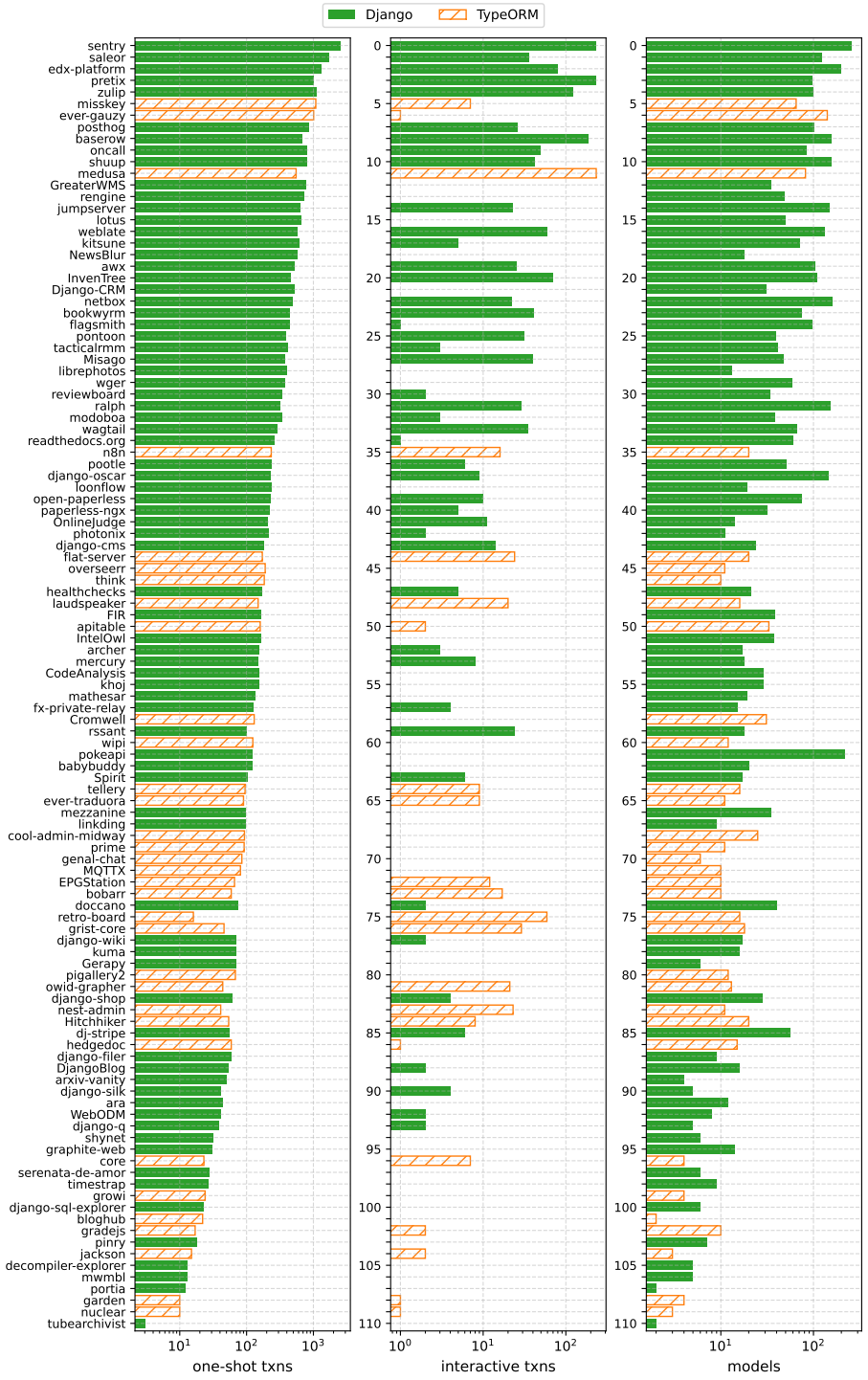

Fig. 2. Number of models and transactions across applications.

popularity, we narrowed this search to the most popular repositories. Specifically, Django repositories were required to have at least 1000 stars, and TypeORM repositories were required to have at least 300 stars. We further refined these sets of applications by removing repositories that only mention the keywords we searched for in their documentation instead of their core codebase, and repositories that are libraries in which ORM usage was only part of their example applications.

Our final corpus consists of 78 Django applications and 33 TypeORM applications[3] that meet these requirements. These applications span a wide range of domains, such as e-commerce, content management, customer relationship management, and blogs. As we exhausted all web applications meeting our criteria on GitHub, we believe that our corpus provides a representative sample for open-source web applications that use ORM to access their data.

Fig. 2 presents the number of models, one-shot transactions, and interactive transactions across the application corpus. The applications are sorted in descending order of the number of transactions (later graphs also present the applications in this order). We discovered 4,778 models and 33,395 transactions[4], including 2,024 interactive transactions (~6% of the transactions). Although most applications have at least one interactive transaction, there are usually one to two orders of magnitude more one-shot transactions than interactive ones per application. The figure also shows that some applications have a very large number of models that can exceed several hundred.

3.2 Semi-automated Annotation

Many of the applications in our corpus have large and complex codebases (hundreds of thousands of lines of code not counting comments[5]), and manually going through each line of code to extract the information we need for our study would be impractical and prone to human error. On the other hand, some of the data we need to extract from these repositories would require complex static analysis algorithms if not done by humans. These algorithms may not consistently yield reliable results across diverse codebases with varying structures, dependencies, and coding conventions.

It therefore become clear that a hybrid mechanism was required in which automated algorithms do the parts of the annotation work that they can reliably complete correctly, while leaving the parts of the codebase that it cannot reliably annotate to a human. To this end, we created a tool to facilitate this hybrid automated/manual annotation. This tool comprises a backend and a frontend.

The backend consists of code tailored to each target programming language. This code parses and traverses the abstract syntax trees (ASTs) of a given application, leveraging the AST-related APIs provided by the *linters* of Python (mypy) and TypeScript (ESLint).

While traversing an AST, the code identifies models and transactions. For example, the Python code identifies models by finding classes that extend the `Model` base class and identifies transactions by matching method calls on objects of the `QuerySet` type to a fixed set of Django operations (e.g., `filter`, `get`, and `update`).

ORM frameworks often use a lazy API, where a query is built by chaining multiple method calls and executed only upon certain *evaluating* methods. For instance, consider the following Django query:

```
Document.objects.filter(type="paper")
                .filter(owner=1)
                .delete()
```

[3]The Django framework, released in 2005, predates TypeORM which was released in 2016. Thus there are more applications adopting Django than TypeORM.
[4]Recall that Section 2 defines a transaction as either an interactive transaction or a one-shot transaction.
[5]For example, the *sentry* application had 740,805 lines of code, *n8n* has 566,146 lines of code, and *saleor* has 485,671 lines of code.

This query calls the `filter` method twice consecutively, but no query is sent to the database server until the `delete` method is called. The `delete` method triggers the construction of a query that deletes all tuples satisfying the conjunction of the two filtering predicates on attributes `type` and `owner`. Since the automation code may mistake these function calls as belonging to separate operations, the frontend facilitates the process of a human following the chain of function calls to consolidate them into a single operation in the official annotation for that repository.

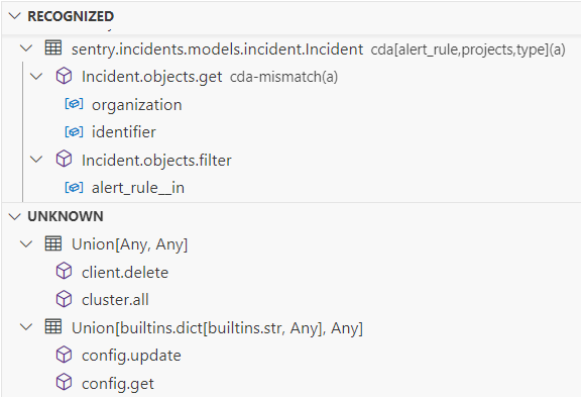

Fig. 3. Screenshot of the data annotation tool. Each list shows the operations and their arguments. The operations are grouped by the types of the objects they are called on.

The frontend invokes the appropriate language-specific code for the opened codebase and displays the results in a graphical user interface, as shown in Fig. 3. Since Python and TypeScript lack strong typing, their type information may be incomplete, leading to the potential for the code to fail to detect all models and transactions without human help. Therefore, the frontend presents two lists: `Recognized` for confidently identified items and `Unknown` for those matched based using simple keyword search, and to be reviewed by a human.

The frontend also includes commands that automates all annotations mentioned in Section 4.1 (i.e. Complex Predicate, CDA Mismatch, and Non-equality Predicate) at operation level, except for detecting attribute dependency in interactive transactions which requires human help.

For interactive transactions, we stated in Section 1 that part of this study involves analyzing whether or not they are convertible to non-interactive transactions. This part of the analysis is more complex and is therefore also done by a human via the frontend.

For example, for the code found in Fig. 1, a human would need to examine the interactive transactions in the `create_document` and `confirm_email_change` functions to determine whether they are Strictly or Non-Strictly Interactive Transactions (Section 5.2), and whether they contain an Attribute Dependency (Section 4.1). Our tool would handle all remaining annotations automatically.

Overall, the tool allows us to focus our manual efforts primarily on interactive transactions, which accounts for approximately 6% of our corpus.

4 Read/Write Set Inferability

Many modern OLTP systems (see Section 1) assume that it is possible to infer the read/write set of a transaction before executing it. These systems use this information to analyze transaction conflicts in advance, and in turn generate potentially more efficient execution schedules. In this section, we study the viability of efficiently extracting the read/write sets from real-world transactions prior to their execution.

4.1 Read/Write Set Definitions

There are multiple ways to define a read/write set (also known as an "access set"), which vary in the precision of what exactly will be read or written. The amount of information available to a system that requires upfront access set knowledge is thus dependent on this definition, which has different implications for implementation complexity and concurrency. The notion of access sets in this section do not refer to the data of a specific database instance. Two transactions conflict if there exists any database instance for which their access sets might overlap. We focus on definitions over the relational data model and SQL, since they are used by the applications in this study.

Definition 4.1 (Relation)**.** Under the Relation definition, the access set of a transaction consists of the union of the set of relations accessed by each statement in the transaction.

The Relation definition is simplest method to define a transaction's access set. SQL queries explicitly specify all accessed relations, making it straightforward to statically extract the access set for each transaction under this definition. However, the lack of precision of what exactly is accessed within that relation forces systems that only have access to this level of granularity to conservatively assume conflicts will occur if two transactions access overlapping relations, even if in reality they will access different sets of tuples. This can result in low concurrency.

At the other extreme[6] is the Predicate definition. Before describing this access set definition, we define a **simple predicate** as a Boolean combination of atomic predicates with the form $attr\ op\ expr$, where $attr$ is an attribute of R, $expr$ is an expression that can be evaluated to a scalar value, and op is one of $\{<, \leq, >, \geq, =, \neq\}$.

Definition 4.2 (Predicate)**.** Under the Predicate definition, the access set of a transaction is the set of triples (R, P, a) obtained over every statement in the transaction. R is a relation name, P is a *simple predicate* selecting tuples from R, and a indicates either a $read$ or $write$ access.

With such information available in advance, potential transaction conflicts can be determined via techniques that work at the predicate level [11]. Two transactions T and T' are considered conflicting if there exist two triples (R, P, a) and (R', P', a') in the access set of T and T', respectively, satisfying all of the following:

(1) $R = R'$
(2) $a = write$ or $a' = write$
(3) There exists a valid instance of R, and a tuple t in that instance such that $P(t) \wedge P'(t) = true$

The Predicate definition allows for detailed knowledge about which tuples will be accessed by a transaction, thereby allowing conflict detection with higher precision, offering the potential for systems to achieve higher concurrency than the Relation definition. Nevertheless, when using the Predicate definition, it is not always possible to infer the complete access set based solely on transaction code (see below). In such cases a system needs to employ **fallback mechanisms**.

One fallback approach is to use the Relation definition for the problematic transactions, effectively assuming they access the entire relation (i.e. $P = true$). An alternative approach is to issue a reconnaissance query that partially executes the transaction code until it can estimate which tuples a transaction will access, such as via the OLLP protocol [44]. However, the transaction may need to be aborted if this estimate is incorrect [39]. Regardless of the fallback mechanism used, they potentially reduce performance of the system and should be minimized.

With the Predicate definition, there are two scenarios in which it is impossible to determine the access set of a transaction directly from the transaction code (thereby guaranteeing the need for a

[6]Technically, this definition can be made more granular by including information about the projected attributes, or even bits within attributes, but we do not consider such definitions in this paper since they are not used in practice.

fallback mechanism): when transactions contain either (1) a complex predicate or (2) an attribute dependency.

Scenario 4.1 (Complex Predicates). A Complex Predicates occurs if a transaction has at least one statement containing a predicate that is *not* a simple predicate.

The Predicate definition we have used thus far is based on previous work that perform transaction conflict detection using predicates [11, 15]. This work restricts such predicates to a simple form in order to make the process of evaluating condition (3) tractable. Otherwise (e.g., if more general forms of predicates had been used) condition (3) is NP-complete. Nonetheless, by restricting predicates to such simple forms, any transactions that use more complex predicates must require fallback techniques.

For example, the second filter operation of the `find_document` function in Fig. 1 contains a complex predicate `endswith` that is true when the suffix of an owner's email matches the given domain.

Scenario 4.2 (Attribute Dependency). An Attribute Dependency arises when a transaction T includes a predicate comparing an attribute with a value that is based on the current state of the data in the database (e.g. the results of a query q).

While T's access set depends on values that must be obtained by performing q, q's results may change over time and thus cannot be certain to be correct unless run in the context of T (and hence are not available in advance of T's execution). Therefore, fallback mechanisms are necessary when a transaction exhibits Attribute Dependencies.

For instance, the `confirm_email_change` function has an Attribute Dependency, where a user is retrieved by their ID originating from a previously retrieved `PendingEmailChange` object.

Use of pure predicate-based concurrency control is rare in industry due to its widely perceived high computational complexity [15]. Instead, many modern systems represent access sets via the set of accessed values of a unique attribute within tuples such as tuple ids or primary keys (i.e., the predicate P only refers to this unique attribute) [30, 44, 50, 51]. For such systems, fallback mechanisms are necessary any time a transaction selects tuples using any other attribute. This approach results in higher concurrency relative to detecting conflict only via the Relation definition, yet also simpler implementations compared to those implemented via the Predicate definition. We refer to the attribute(s) used for conflict detection in such systems as the **conflict detection attribute(s)** (CDA). The CDA of a relation is an ordered subset of the attributes of the relation and can be defined on relations that do not have primary keys. Each relation has exactly one CDA.

To determine the **selected CDAs** of a statement S accessing a single relation R with a *simple predicate* P, we first transform P into its disjunctive normal form $c_1 \vee c_2 \vee \ldots \vee c_n$. Let $R(c)$ be the set of R's attributes in c. The selected CDAs of S is the set $\{(R(c_1), R(c_2), \ldots, R(c_n)\}$.

If S involves a join, we perform the following transformation performed by almost every query optimizer[7].

$$\sigma_\beta(R_1 \bowtie_\theta R_2) \equiv \sigma_{\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n}(R_1 \bowtie_\theta R_2) \quad (1)$$

$$\equiv \sigma_{\gamma_1 \wedge \gamma_2 \wedge \gamma_3}(R_1 \bowtie_\theta R_2) \quad (2)$$

$$\equiv \sigma_{\gamma_3}(\sigma_{\gamma_1}(R_1) \bowtie_\theta \sigma_{\gamma_2}(R_2)) \quad (3)$$

In equivalence (1), β is transformed into its conjunctive normal form $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n$. In (2), these β clauses are sorted into three groups γ_1, γ_2, and γ_3 such that γ_1 contains clauses that only refer to R_1, γ_2 contains clauses that only refer to R_2, and γ_3 contains clauses that refer to both R_1

[7]Projection is omitted as it is irrelevant

and R_2. Finally, in (3), we use the selection distribution equivalence rule [23] to push the γ_1 and γ_2 predicates down to R_1 and R_2. γ_1 and γ_2 thus specify the attributes of R_1 and R_2. We then use the aforementioned steps to obtain the CDAs of S over both R_1 and R_2 from γ_1 and γ_2, respectively.

Definition 4.3 (CDA-Based)**.** Under the CDA-Based definition, the access set of a transaction is the union of the *selected CDA* sets of all statements in the transaction.

The CDA-Based definition is only valid for transactions using simple predicates, so it shares the same fallback scenarios as the Predicate definition.

Systems that detect conflict using the CDA-Based definition often create an index on the CDA of the relation so that locking-based mechanisms can be used directly on index entries. To use this index, the selected CDA must form a *prefix* of the relation's CDA. For example, the CDA of the `Document` relation in Fig. 1 is the ordered subset `(type, create_date)`. If a transaction selects tuples in this relation with the predicate `type = "paper"`, the system can lock all entries in the CDA index satisfying that predicate. Subsequent transactions selecting with `type = "paper"` and optionally an arbitrary value of `create_date` will be detected as conflicting with the previous transaction. As a result, the CDA-Based definition has the following additional fallback scenario.

Scenario 4.3 (CDA Mismatch). A CDA Mismatch occurs when a transaction has a selected CDA $R(c)$ that does not form a prefix of R's CDA.

In Fig. 1, the CDA of each model is represented by the shaded columns. The `create_document` and `confirm_email_change` functions involve user selection based on their email addresses rather than the attribute `id`, resulting in a CDA Mismatch. Additionally, the last line of the `find_document` function in Fig. 1 is a join between the `Document` and the `User` relations. It does not exhibit CDA Mismatch with respect to `Document` since it accesses the CDA prefix `(type, create_date)`, whereas it exhibits CDA Mismatch with respect to `User` since it accesses the attribute `email`, which is not a CDA prefix of `User`. Common sources of CDA Mismatches are transactions that perform lookups in non-CDA indexes (e.g., secondary indexes when the CDA is the primary key).

The CDA-Based definition, as stated above, allows both equality and non-equality predicates on the CDAs. In practice, some systems do not detect conflicts via arbitrary predicates but instead detect conflicts on individual tuples via references to those tuples' CDA. This avoids the need for the advanced data structures and optimizations necessary to track predicates, but may result in poor performance for range queries (e.g., those involving $<$, $>$, $\neq$ operators). In order to investigate the impact of this simplification on the applications in our corpus, we introduce the following definition.

Definition 4.4 (CDAeq-Based)**.** The CDAeq-Based definition is the same as the CDA-Based definition except that it is not valid for transactions involving non-equality operations in their predicates.

A fallback is thus necessary in the following scenario.

Scenario 4.4 (Non-equality Predicates). This scenario arises if a transaction includes at least one non-equality operation in its predicates.

In our running example, the `find_document` function selects documents by a range of their creation dates and thus requires a fallback using the CDAeq-Based definition. Although all complex predicates can be qualified as non-equality predicates, we do not count them in this category to distinguish between simple non-equality predicates and complex predicates in our dataset.

The four different access-set definitions we have defined in this section are summarized in Fig. 4. Implementation complexity decreases from left to right on the chart. When implementation

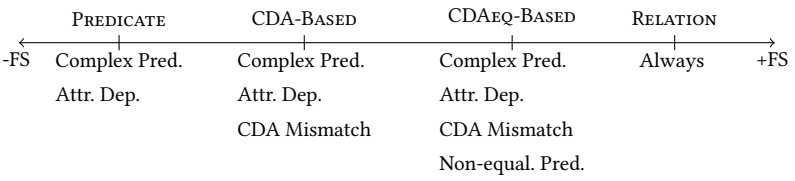

Fig. 4. Access set definitions and their fallback scenarios

complexity decreases, so does the precision of conflict detection, so the likelihood of false conflicts and the frequency of fallbacks increase from left to right. Reduced isolation levels eliminate or reduce the time duration of conflicts and thus are less impacted by the false conflicts introduced by fallbacks. Nonetheless, except for extremely low levels (e.g. READ UNCOMMITTED which does not check for conflicts at all), they too experience reduced performance as fallback frequency increase. However, it is important to note that the need to use fallbacks, or the lack thereof, does not solely determine the overall performance of the system. Implementation overhead, such as that of evaluating the satisfiability of boolean expressions when using the Predicate definition, can significantly affect performance. It is important to find a balance between implementation overhead of a particular access-set definition and frequency of its fallbacks. As the research community continues to develop new algorithms, they will evaluate the overhead of their own implementations, but they will make assumptions about the frequency of fallbacks. The purpose of our investigation is to provide some data to quantify these assumptions.

4.2 Analysis of our Application Corpus

In this section, we summarize the results of our analysis of the application corpus regarding the inferability of access sets in advance of transactional execution. For each definition of an access set discussed in the previous section and summarized in Fig. 4, we annotate each transaction in each application in the corpus to indicate if it will require fallback mechanisms using that definition. This section presents aggregate data on this analysis.

Two of our definitions, CDA-Based (Definition 4.3) and CDAeq-Based (Definition 4.4), are based on the concept of a CDA, which is not something declared explicitly in application code. Therefore, we assigned each relation a CDA based on the frequency of transactions that access data from that relation by individual attributes. Whichever attribute(s) are used most frequently (to specify which tuples within that relation that a transaction should access) is chosen as the CDA. CDA Mismatches (Scenario 4.3) are then annotated for any transaction that looks up data by a different attribute for that relation. For example, if the only operations on the `User` relation were those shown in Fig. 1, (`email`) would be assigned as the CDA for `User`, since it would result in two transactions matching the CDA and only one CDA Mismatch. However, in the figure, (`id`) is shown as being the CDA since in practice there would be more transactions than those shown in the figure for this application, and it is likely that (`id`) would be the most frequent look-up attribute when considering the entirety of these additional transactions.

After assigning CDAs, we annotated each transaction with one or more scenarios described in Section 4.1 (Scenario 4.1-4.4). Finally, for each access set definition (Definition 4.1-4.4), we determine if a transaction requires a fallback based on the chart in Fig. 4.

Fig. 5 shows the frequencies of different fallback scenarios in the application corpus. Each bar from left to right is associated with an individual application in the corpus, and the height of each bar corresponds to the percentage of transactions within that application that would result in that fallback scenario when using access-set definitions for which that fallback scenario is possible.

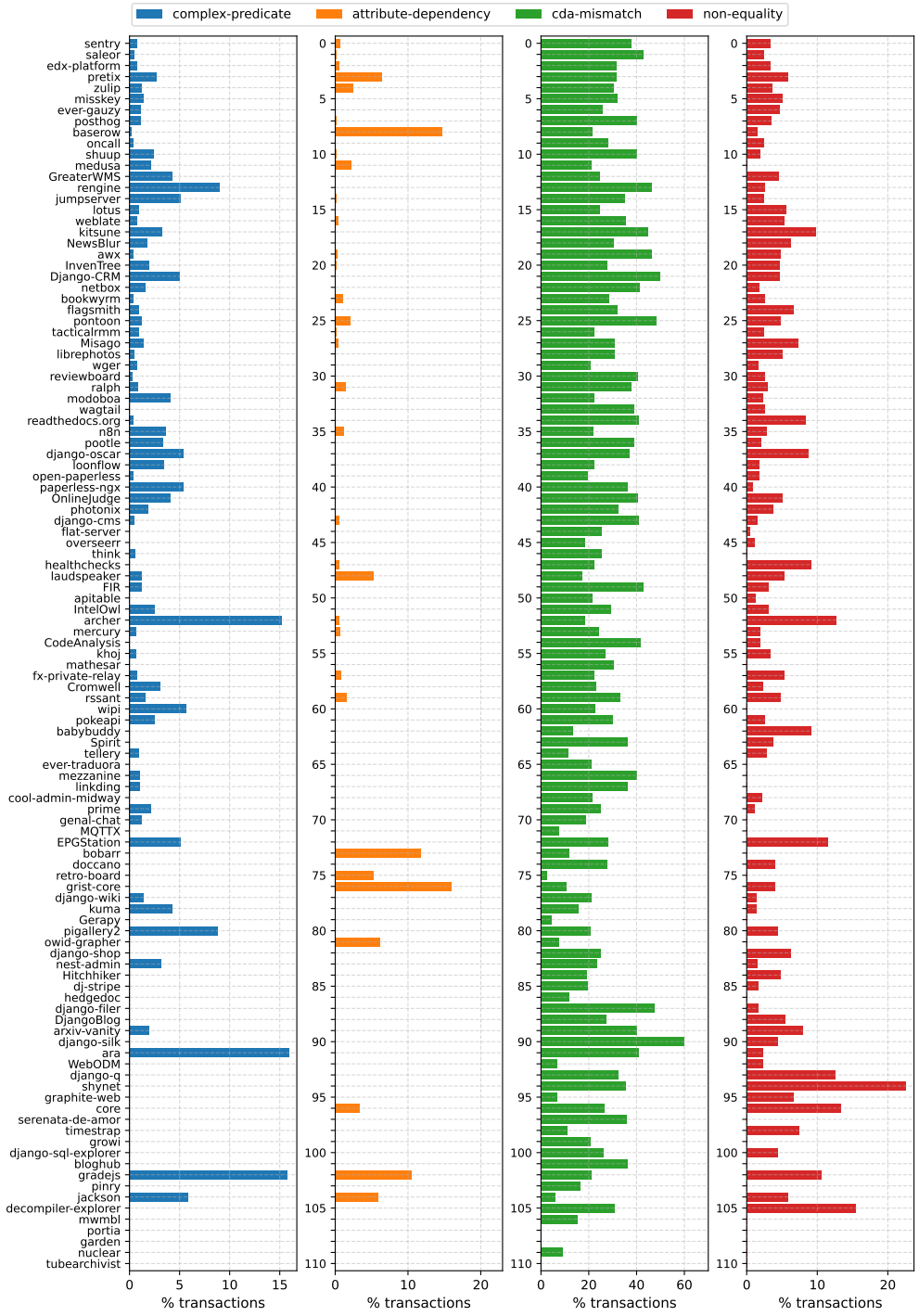

Fig. 5. Percentages of the fallback scenarios.

Nearly all applications have at least one CDA Mismatch (108/111 ≈ 97%) and Non-Equality Predicate (89/111 ≈ 80%) transaction. On average, 26.9% transactions per application would result in a CDA Mismatch, and 3.7% of transactions per application have (non-complex) Non-Equality Predicates. In contrast, complex predicates and attribute dependencies are less common. Complex predicates only exist in (69/111 ≈ 62%) of the applications, averaging 1.7% of all transactions per application among the 69 applications. Attribute dependencies only exist in (33/111 ≈ 30%) of the applications, averaging 0.9% of transactions per application among the 33 applications. The bulk of complex predicates involve string pattern matching (e.g., LIKE), while the remaining few are predicates on non-scalar fields, such as checking whether an array contains a given value or whether a JSON object has a specific key.

Using these annotations, we can quantify the need for fallback mechanisms for the different access-set definitions. For each definition, we count the number of transactions containing one of the annotations from the corresponding column in Fig. 4. We then compute the percentage of such transactions for every application. The Relation definition (Definition 4.1) is equivalent to performing fallbacks for all transactions; we include it here for completeness.

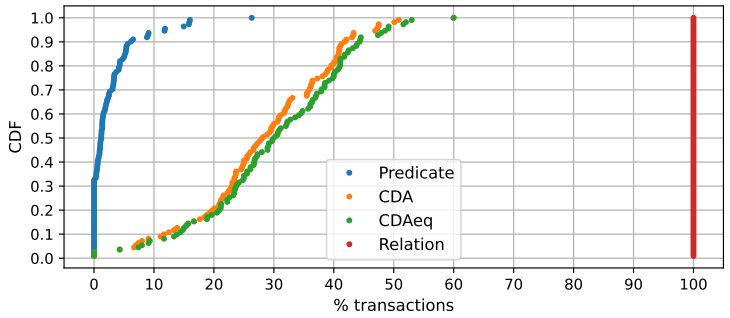

Fig. 6. CDF of transaction percentages requiring fallbacks for each read/write set definition.

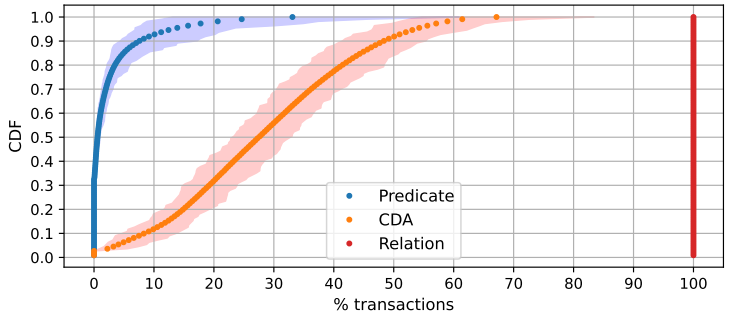

Fig. 7. CDF of transaction percentages requiring fallbacks for each read/write set definition in the Zipf-distributed workload simulation.

The fallback percentages for each access-set definition are presented as cumulative distribution function (CDF) curves in Fig. 6. Each point on a curve corresponds to an application, showing the percentage of transactions within that application that require a fallback. From left to right, the

curves follow the same order as in Fig. 4 since the percentage of transactions requiring fallbacks logically increases with the number of possible scenarios that trigger the fallbacks.

Under the CDA-Based definition (Definition 4.3), only 3% of applications requiring no fallbacks, but over 90% of applications can run the majority of their transactions without fallbacks. The curve of CDAeq-Based definition (Definition 4.4) is shifted only slightly to the right of the CDA-Based curve, since the use of non-equality predicates were not common in our corpus. We will further discuss these results in Section 6.

Non-uniform transaction frequency. In our analysis thus far, we have assumed a uniform distribution of transaction frequencies in each application's workload. However, real-world workloads often exhibit non-uniform distributions. Since we lack access to the actual production workloads of the applications in our corpus, we explore how our analysis behaves under a non-uniform workload by modeling transaction frequencies using a Zipf distribution with the probability mass function given by

$$P(i; s, N) = \frac{1/i^s}{\sum_{j=1}^{N} 1/j^s}$$

where i is the rank of a transaction, s is the skewness parameter, N is the total number of transactions, and $\sum_{j=1}^{N} 1/j^s$ is the normalization factor. For every application, we randomly assign the ranks to the transactions, compute their frequencies using the above formula at $s = 1.0$, and redo the above analysis with each transaction weighted by their simulated frequency. We repeat this process 1000 times, each with a different rank assignment. Fig. 7 presents the simulation results. For each read/write set definition, the shaded region covers the minimum to maximum percentages of transactions requiring fallbacks across all simulated runs, while the middle curve indicates the mean. The result for CDAeq-Based closely resembles that of CDA-Based and is slightly shifted to the right with respect to CDA (as observed in Fig. 6); therefore it is not shown on the graph (to keep the graph easy to read). Overall, the means of simulation result follows closely with the curves of the uniformly distributed workload and the variations in the curves remain within an acceptable range.

5 Transaction Interactivity

Many modern OLTP systems target applications that do not require the flexibility of interactive interfaces. These systems restrict applications to sending each transaction to the system in a single round-trip, rather than allowing multiple round-trips within a single transaction. This section discusses the advantages and disadvantages of such restrictions, and presents results on whether the applications in our corpus require the flexibility of interactive interfaces in their current and potential modified states.

5.1 One-shot vs. Interactive Transactions

One-shot transactions are implemented either as precompiled stored procedures that are invoked from the client, or as code snippets sent to the database server for on-the-fly interpretation, or as singleton SQL commands. These transactions generally offer better performance for two reasons. First, if implemented as precompiled stored procedures, the SQL parsing and planning steps are done only once and can be skipped in subsequent invocations. Second, they reduce communication overhead by requiring only one message round-trip, allowing the database system to focus on processing the transaction instead of allocating excessive processing cycles to the client communications process and paying the latency cost of waiting and deserializing commands from the client. Additionally, one-shot transactions enable new techniques that plan entire groups of

transactions in advance, allowing for generation of more efficient execution schedules. For example, Janus [30], Aria [27], Detock [31], and Caerus [17] reorder the transactions to avoid aborts, and Strife [33] clusters the transactions such that they can be executed in parallel without concurrency control.

However, one-shot transactions can add complexity to writing and maintaining application code. Instead of interleaving database access syntax with application logic inside the application codebase, developers must write application and database access logic as separate code units. For database access logic, they must use a programming language supported by the database server, which may differ from the primary language used for the application. This requires developers to be familiar with additional database-specific languages and features. Moreover, this separation between application and transaction code prevents transactions from performing tasks typically handled within the application process, such as accessing files on the application server or calling external APIs.

5.2 Annotations & Results

In addition to determining whether a transaction is interactive, we aim to assess, for each application in the corpus, the feasibility of converting interactive transactions into one-shot transactions. We define an **external operation** as any operation that retrieves data from or induces a side effect outside the current process. Examples include file system actions such as creating or deleting files, making API calls to other services over the network, or performing inter-process communication. Such operations significantly complicate the process of converting interactive transactions into one-shot transactions, because they may be impossible, hard, or unwise to perform inside the database system. We therefore annotate applications in our corpus with such operations according the methodology we describe below.

Definition 5.1 (Strictly Interactive Transactions)**.** We annotate a transaction as strictly interactive if it satisfies all of the following conditions:

(1) It contains at least one external operation.
(2) The invocation of the external operation depends on the result of a database operation.
(3) The result of the external operation influences a database operation, e.g., via changing the control flow (i.e. if-then, for-loop) that decides how many times or whether the database operation is called, or is used as an input to the database operation (possibly after some transformations).

The `create_document` function in Fig. 1 serves as an example of a strictly interactive transaction. It contains two external operations: `path.exists`, which checks for the existence of a file, and `write_file`, which creates a file at a specified path (therefore it meets condition (1) above). The file path is constructed using a user ID from a tuple read from the database (condition (2)). The result of the file existence check determines whether a new document is inserted into the `Document` relation (condition (3)). Although it may appear that there is no dependency between the `write_file` invocation and the insertion into the `Document` relation, if the `write_file` call fails (e.g., due to lack of write permission), the insertion will be rolled back. Therefore, a dependency exists between them.

When there are no dependencies between database state and external operations, or no dependencies between external operations and the actions of a transaction, it is often straightforward to rewrite transactions that include external operations to an equivalent transaction where the operations are removed and performed outside of transactional boundaries before or after the database request (see below for an example). In contrast, when these dependencies exist, such rewrites would require deeper semantic knowledge of the application and are, in many cases, impossible.

Some external operations can be supported by database systems via plugin mechanisms. However, such mechanisms require extra time and effort to develop and maintain. Furthermore, they shift additional responsibilities to the database system, which could fundamentally change the organization of the entire application stack and introduce new security concerns. For example, if the `create_document` function is to be executed as a one-shot transaction within the database system, the file must reside on the same server running the database system, or the database server must access the file through a network file system or blob storage. From a software engineering perspective, this undermines the separation of concerns [49] between database systems, which manage structured data, and the components that handle blob data. Moreover, for a database system to access external APIs, it must be granted more permissions (e.g., via API keys or tokens) that it normally would not have, thereby increasing the attack surface and complicating security access management.

Definition 5.2 (Non-strictly Interactive Transaction)**.** We annotate a transaction as non-strictly interactive if it does not satisfy at least one of the above conditions, since it is more amenable to conversion into one-shot transactions. The function `confirm_email_change`, for example, is not strictly interactive because condition (1) is false (it does not contain any external operations).

If neither condition (2) nor (3) is satisfied, it does not matter whether the external operations are performed inside or outside the transaction because their side effects are neither managed nor automatically rolled back by the database system. For instance, if the `path` variable in the `create_document` function depended on the user's email instead of the user's ID, neither the `path.exists` nor the `write_file` call would depend on the `User.get` operation, failing condition (2). Consequently, the transaction would be annotated as non-strictly interactive as it could be rewritten with the file system accesses moved outside the transaction as follows:

```
def create_document(email, type, root_dir, name, content):
  path = root_dir / email / name
  if not path.exists():
    write_file(path, content)
    with atomic(): # the transaction starts here
      user = User.get(email=email)
      Document.create(type=type, create_date=today(),
                      path=path, owner=user.id)
```

With this rewrite, the file operations can be performed on the client side and the transaction portion contains only database operations. The transaction portion can thus be sent to the database system as a one-shot transaction.

The left most plot in Fig. 8 shows the number of one-shot, strictly interactive, and non-strictly interactive transactions for each application in our corpus. Out of the 111 applications, 68 of them (~61%) contain interactive transactions (of either type), and 22 of them (~20%) have strictly interactive transactions. Among the 68 applications with interactive transactions, an average of 9.6% of transactions per application are interactive. For the 22 applications with strictly interactive transactions, an average of 2.5% of transactions per application are strictly interactive (or 0.5% of transactions per application when accounting across all applications). The *retro-board* application is the only one where interactive transactions (~79%) outnumber one-shot transactions (~21%); yet all its interactive transactions can be converted into one-shot transactions.

After diving deeper into the characteristics of strictly interactive transactions, we observed that while an application may have multiple strictly interactive transactions, it is either dominated by or has only one type of external operation that gives rise to its strictly interactive transactions. For example, if an application contains a strictly interactive transactions involving calling an external API, it is almost always the case that other strictly interactive transactions call that same API. Only

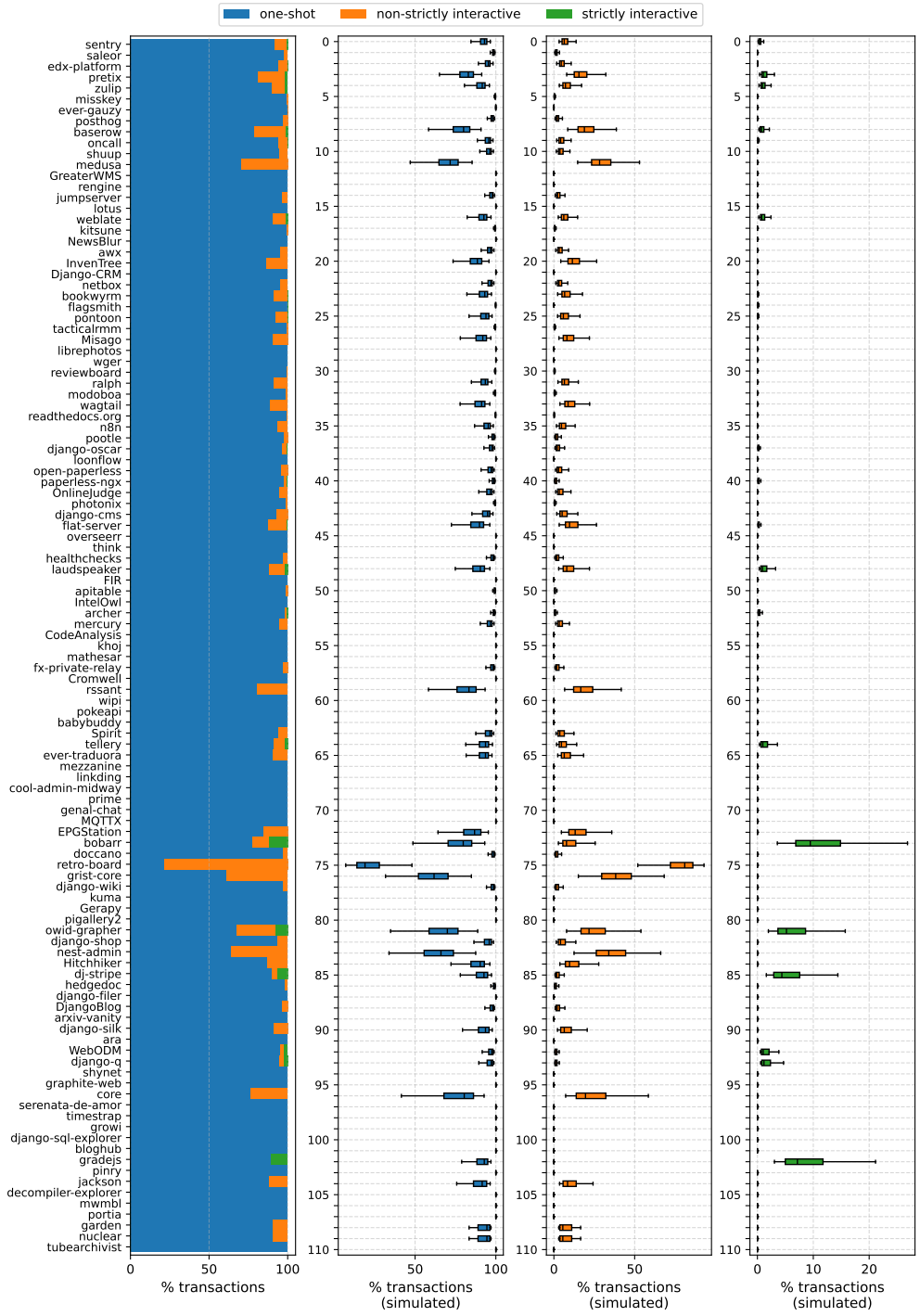

Fig. 8. Transaction interactivity (leftmost) and transaction interactivity in the Zipf-distributed workload simulation.

four applications had more than one type of external operation (*pretix*, *zulip*, *baserow*, and *bobarr*), and even within these four applications, only 22% of external operations were different from the most common type. Table 1 summarizes the percentage of strictly interactive transactions in each of the 22 applications that have strictly interactive transactions, and describes the most common external operation for each one.

We did not observe any differences with respect to the ORM framework. The proportion of applications with strictly interactive transactions for the Django applications was ($16/78 \approx 20\%$) and for TypeORM was ($6/33 \approx 18\%$). Although strictly interactive transactions are rare, the types of external operations involved are diverse, with no single type dominating the majority of applications.

The applications in Table 1 are sorted in descending order of the number of their transactions. The table thus indicates that strictly interactive transactions tend to account for a larger proportion in applications with fewer operations. We hypothesize that this is because external operations are typically limited in scope. For instance, the number of API endpoints are usually fixed and minimized for easy management and to reduce the attack surface. External operations also need to be called only in a few specialized transactions (e.g., not all transactions send an email). Consequently, as an application scales, the number of transactions with external operations remains relatively constant, leading to a smaller proportion of these transactions in larger applications.

Finally, we identified 9 applications with interactive transactions that involve external operations but do *not* qualify as strictly interactive transactions. Among them, 6 applications (*archer*, *flat-server*, *OnlineJudge*, *n8n*, *ralph*, and *kitsune*) contain transactions that do not meet condition (2), while 3 others (*paperless-ngx*, *weblate*, and *pontoon*) contain transactions that do not meet condition (3).

Table 1. Applications with Strictly Interactive Transactions (SIT). TypeORM applications are marked with an asterisk (*).

Application	% SIT	Description of the external operations
sentry	0.68	Making remote procedure calls
edx-platform	0.07	Sending emails
pretix	3.90	Sending emails
zulip	1.49	Sending data to message queues
baserow	1.39	Reading files
oncall	0.23	Sending data to message queues
weblate	1.41	Performing version control commit
bookwyrm	0.21	Making HTTP requests
flagsmith	0.23	Calling DynamoDB API
pontoon	0.24	Performing version control commit
django-oscar	0.42	Creating files
paperless-ngx	0.45	Creating/deleting files
flat-server*	0.51	Calling Redis API
laudspeaker*	1.78	Sending data to message queues
archer	0.63	Sending emails
tellery*	1.92	Sending emails
bobarr*	11.8	Creating/deleting files and directories
owid-grapher*	7.69	Appending data to files
dj-stripe	6.56	Calling Stripe API
WebODM	2.33	Copying files
django-q	2.5	Sending data to message queues
gradejs*	10.5	Calling Amazon S3 API

Non-uniform transaction frequency. To understand how our analysis varies under non-uniform workloads, we simulate transaction frequencies using the Zipf distribution similar to the process described in Section 4.2. The results, shown in the three rightmost plots in Fig. 8, use a box plot to summarize the percentage of each type of transaction interactivity per application. The presence of skew causes strictly-interactive transactions to become a larger issue for 4 applications (*bobarr*, *owid-grapher*, *dj-stripe*, and *gradejs*) if the frequency distribution favors these transactions. In addition, there are 5 more applications where the skewed frequency may cause interactive transactions may surpass one-shot transactions (*medusa*, *grist-core*, *owid-grapher*, *nest-admin*, and *core*). Overall, we believe that the discussion in Section 6 remains broadly applicable to skewed workloads.

6 Discussion

In this section, we discuss the implications that can be drawn from the data presented in the previous sections for existing OLTP systems and suggest future research directions.

6.1 Read/Write Set Inferability

The data in Section 4 paints a nuanced picture for systems that require upfront knowledge about what data a transaction will access. For 90% of applications, at least 58% of their transactions avoid the need for fallback mechanisms. While the proportion of transactions requiring fallbacks is not insignificant, they are primarily due to CDA Mismatches (Scenario 4.3) . Unlike fallbacks arising from Attribute Dependency (Scenario 4.2), which may involve analyzing chains of dependencies in multi-statement transactions and performing multiple OLLP lookups, inferring the access set for CDA Mismatches is simpler. It can be achieved by reading a non-CDA index to reveal the CDA of the selected tuples, resulting in a more lightweight fallback technique. Moreover, CDA Mismatches mostly occur in single-statement transactions, meaning the windows between optimistic reads and their validation are much shorter, reducing the likelihood of aborts in OLLP-type mechanisms.

These results also indicate that future research on improving OLLP or designing new fallback techniques should focus on minimizing CDA Mismatch frequency. One approach could involve moving towards a predicate-based definition of an access set, potentially implemented through predicate locking. Gaffney et. al [15] note that predicate locking is often overlooked by the research community due to its perceived high overhead. Yet, they demonstrate that it can be a viable concurrency control method with several optimizations that significantly enhance its performance and scalability. They further use predicate locking as a basis for modularizing transaction isolation into a separate service. Such modularization aligns well with the design of deterministic database systems [2], which frequently require *a priori* knowledge of access sets. As a first step, future research can explore how to modify the deterministic lock manager [31, 38, 44] to support predicate locking along with the optimizations introduced in [15].

Lastly, the small gap in the results between CDA-Based (Definition 4.3) and CDAeq-Based (Definition 4.4) definitions suggests that it would be reasonable for existing and future systems to initially support only equality predicates and use fallbacks when it comes to non-equality predicates. Nonetheless, most applications (~80%) contain at least one non-equality predicate, so the long term plan for such systems should likely include support for non-equality predicates as well.

6.2 Transaction Interactivity

The data in Section 5 supports the notion that OLTP systems which do not support interactive transactions can still cater to a wide range of applications. Specifically, 39% of the applications in our corpus do not use interactive transactions at all. The remaining 61% of the applications, without further modifications, are unable to use any OLTP systems that support only one-shot transactions

(referred to as *one-shot-only systems* in this section). However, as noted in Section 5, many of these applications can be rewritten in a straightforward manner to use only one-shot transactions, and thus can use one-shot-only systems with reasonable rewrite effort.

One research direction is to embrace interactive transactions and extend the protocols of the one-shot-only systems to accommodate them. Recall that among applications that contain interactive transactions, only an average of 9.6% of their transactions are interactive. This suggests that interactive transactions likely constitute only a small portion of the live workload. Therefore, the extended protocols may not need to strive for optimal performance for interactive transactions, but rather focus on *enabling* them while maintaining the original advantages of one-shot transactions.

Another research direction is leveraging the fact that the vast majority of interactive transactions can be rewritten as one-shot transactions. Previous work has indicated that applications seldom use stored procedures because they introduce other challenges in the development process [18, 32]. Although significant research has focused on improving the performance of executing stored procedures, their usability has not advanced much, particularly in areas like versioning, debugging, and testing [16]. Hence, more research is needed to reduce the friction associated with employing them. One potential solution is to automate the synthesis of stored procedures from interactive transactions using static analysis, as proposed in WeBridge [18].

6.3 Limitations

This section outlines some limitations of our study.

Firstly, by extracting code from GitHub for the application corpus, we only have access to the transaction code itself, not the actual composition of the application workload and how often particular transactions are executed. Nonetheless, the simulations in Section 4.2 and Section 5.2 provide some evidence that our analysis is robust to workload skewness.

Secondly, as discussed in Section 5.1, one-shot transactions can be implemented as precompiled stored procedures, code snippets, or singleton SQL commands. Our corpus includes only one-shot transactions that have the form of singleton SQL commands.

Thirdly, for Strictly Interactive Transactions (Definition 5.1), certain real-world scenarios exist where external operations, even if they do not directly influence database operations, cannot be moved outside of transaction boundaries. For example, a transaction may include observability code that logs read data to an external service. While this read data could theoretically be passed back to the client for logging after the transaction, if the transaction is aborted, this approach risks losing partially logged information that could be used to diagnose the cause of abort. Since aborts are rare in practice, we do not consider such scenarios. Therefore, our Strictly Interactive Transaction result may slightly underestimate the actual percentage.

Finally, Bailis et. al [5] and Tang et. al [42] suggest that applications using ORM frameworks often rely on application-level mechanisms for data integrity and concurrency control, rather than utilizing equivalent features supported by the database system. Our analysis do not capture these usage patterns.

7 Related Work

Application studies. Other studies on application transactions in real-world applications have been performed in other contexts. Bailis et. al [5] investigate concurrency control mechanisms that are implemented at application-level—or *feral* concurrency control—in 67 Ruby on Rails applications. Warszawski and Bailis [48] analyze traces of database activity and identify security vulnerabilities stemming from weak isolation levels in 22 e-commerce applications. Tang et. al [42] study the characteristics, correctness, and performance of 91 *ad hoc* transactions among 8 web applications. Cheng et. al [7] examine 93 real-world concurrency bugs in database-backed

applications to understand their root causes, consequences, and their fixes. Our work is orthogonal to these studies and provides comprehensive data on the interactivity and access set inferability aspects of transactions in real-world applications.

Read/write set inference. While conflict detection at relation level often results in a low degree of parallelism, IC3 [47] demonstrates that this information—easily obtained from the transaction code—can be leveraged to extract more concurrency in workloads whose transactions frequently access multiple relations at once. As mentioned in Section 4.1, systems relying on more fine-grained definitions of the read/write set, such as Predicate (Definition 4.2) or CDA-Based (Definition 4.3), may need to utilize OLLP [44]. Prognosticator [20] refines OLLP further by using symbolic execution to narrow down portion of the the read/write set that requires a reconnaissance query.

One-shot transactions. One approach to writing one-shot transactions is through specialized APIs. Sinfonia [3] introduces an API called *minitransactions*, each of which may contain operations such as comparing, reading, and writing specified data items. Similarly, Amazon DynamoDB [19] offers 3 transactional operations: `CheckItem`, `TransactGetItems`, and `TransactWriteItems`. Fauna [1] provides its own query language, FQL, inspired by TypeScript, that allows more expressive transactions with features like conditional branching (i.e., if-else). However, these APIs often come with limitations in expressiveness or require developers to learn new syntax. To avoid these issues, Thomson et. al [45] suggest a technique where developers write code snippets in a general-purpose programming language (e.g., Python) and submit them to the database system, which interprets the code on-the-fly.

Another approach to writing one-shot transactions is to use stored procedures, which are employed by nearly all systems discussed in Section 1. However, Pavlo [32] found that most production database rarely use stored procedures, mainly because they pose challenges in software developments (e.g., DBAs are often reluctant to update them frequently). To address these challenges, WeBridge [18] proposes using static analysis to automate the generation of stored procedures from web application source code. Similarly, Apiary [24] allows developers to write functions in Java and use SQL to access data, then compiles these functions into stored procedures.

8 Conclusion

This empirical study investigated the prevalence of design assumptions underlying modern OLTP systems, specifically the trade-offs between transaction interactivity, access set inferability and efficiency. Our comprehensive analysis of open-source applications reveals that while access set inferability is achievable for a substantial number of transactions, fallback mechanisms are still necessary for many others, but can be designed to be lightweight. Furthermore, a significant proportion of applications can operate effectively without interactive transactions, and many existing interactive transactions can be converted to one-shot transactions with minimal modifications.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions that helped us improve the quality of this paper. This work is supported by the National Science Foundation under grants DGE-1840340 and IIS-1910613.

References

[1] [n. d.]. Fauna: Architectural overview. https://assets.ctfassets.net/po4qc9xpmpuh/2LkoSujDTxtMWdzso84WDw/0abe5c3a121b83096ab99830c01e147a/Fauna_architectural_overview.pdf. Accessed: 2024-7-28.

[2] Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (Aug. 2018), 78–88. doi:10.1145/3181853

[3] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating*

systems principles (SOSP '07, Vol. 41). Association for Computing Machinery, New York, NY, USA, 159–174. doi:10.1145/1294261.1294278

[4] M M Astrahan, M W Blasgen, D D Chamberlin, K P Eswaran, J N Gray, P P Griffiths, W F King, R A Lorie, P R McJones, J W Mehl, G R Putzolu, I L Traiger, B W Wade, and V Watson. 1976. System R: relational approach to database management. *ACM Trans. Database Syst.* 1, 2 (June 1976), 97–137. doi:10.1145/320455.320457

[5] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1327–1342. doi:10.1145/2723372.2737784

[6] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 210–227. doi:10.1145/3447786.3456238

[7] Chao-Wei Cheng, Mingzhe Han, Nuo Xu, Spyros Blanas, Michael D Bond, and Yang Wang. 2023. Developer's responsibility or database's responsibility? Rethinking concurrency control in databases. *CIDR* (2023).

[8] James Cowling and Barbara Liskov. 2012. Granola:{Low-Overhead} distributed transaction coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 223–235.

[9] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proceedings VLDB Endowment* 16, 8 (April 2023), 1930–1943. doi:10.14778/3594512.3594523

[10] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient replication via timestamp stability. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 178–193. doi:10.1145/3447786.3456236

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

[12] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings VLDB Endowment* 8, 11 (July 2015), 1190–1201. doi:10.14778/2809974.2809981

[13] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings VLDB Endowment* 10, 5 (Jan. 2017), 613–624. doi:10.14778/3055540.3055553

[14] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings VLDB Endowment* 12, 11 (July 2019), 1471–1484. doi:10.14778/3342263.3342627

[15] Kevin P Gaffney, Robert Claus, and Jignesh M Patel. 2021. Database isolation by scheduling. *Proceedings VLDB Endowment* 14, 9 (May 2021), 1467–1480. doi:10.14778/3461535.3461537

[16] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural extensions of SQL: understanding their usage in the wild. *Proceedings VLDB Endowment* 14, 8 (April 2021), 1378–1391. doi:10.14778/3457390.3457402

[17] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2024. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proceedings VLDB Endowment* 17, 3 (Jan. 2024), 469–482. doi:10.14778/3632093.3632109

[18] Gansen Hu, Zhaoguo Wang, Chuzhe Tang, Jiahuan Shen, Zhiyuan Dong, Sheng Yao, and Haibo Chen. 2024. WeBridge: Synthesizing Stored Procedures for Large-Scale Real-World Web Applications. *Proc. ACM SIGMOD Int. Conf. Manag. Data* 2, 1 (March 2024), 1–29. doi:10.1145/3639319

[19] Joseph Idziorek, Alex Keyes, Colin Lazier, Somu Perianayagam, Prithvi Ramanathan, J C Sorenson, D Terry, and Akshat Vig. 2023. Distributed transactions at scale in Amazon DynamoDB. *Proc. USENIX Annu. Tech. Conf.* (2023), 705–717.

[20] Shady Issa, Miguel Viegas, Pedro Raminhas, Nuno Machado, Miguel Matos, and Paolo Romano. 2020. Exploiting symbolic execution to accelerate deterministic databases. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, Vol. 00. IEEE, 678–688. doi:10.1109/icdcs47774.2020.00040

[21] Evan P C Jones, Daniel J Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 603–614. doi:10.1145/1807167.1807233

[22] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P C Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings VLDB Endowment* 1, 2 (Aug. 2008), 1496–1499. doi:10.14778/1454159.1454211

[23] Henry F Korth, S Sudarshan, and Abraham Silberschatz. 2019. Query Optimization. In *Database System Concepts* (6 ed.). McGraw-Hill Education.

[24] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, Peter Bailis, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2022. Apiary: A DBMS-Integrated

Transactional Function-as-a-Service Framework. *arXiv [cs.DB]* (Aug. 2022). arXiv:2208.13068 [cs.DB]

[25] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When Private Blockchain Meets Deterministic Database. *Proc. ACM SIGMOD Int. Conf. Manag. Data* 1, 1 (May 2023), 1–28. doi:10.1145/3588952

[26] Si Liu, Luca Multazzu, Hengfeng Wei, and David A Basin. 2024. NOC-NOC: Towards Performance-optimal Distributed Transactions. *Proc. ACM SIGMOD Int. Conf. Manag. Data* 2, 1 (March 2024), 1–25. doi:10.1145/3639264

[27] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings VLDB Endowment* 13, 12 (Aug. 2020), 2047–2060. doi:10.14778/3407790.3407808

[28] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based commit and replication in distributed OLTP databases. *Proceedings VLDB Endowment* 14, 5 (Jan. 2021), 743–756. doi:10.14778/3446095.3446098

[29] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: scaling transactions through asymmetric replication. *Proc. VLDB Endow.* 12, 11 (July 2019), 1316–1329. doi:10.14778/3342263.3342270

[30] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 517–532. doi:10.5555/3026877.3026917

[31] Cuong D T Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proc. ACM SIGMOD Int. Conf. Manag. Data* 1, 2 (June 2023), 1–27. doi:10.1145/3589293

[32] Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 3. doi:10.1145/3035918.3056096

[33] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 527–542. doi:10.1145/3318464.3389764

[34] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *EDBT*. 73–84. doi:10.5441/002/edbt.2020.08

[35] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 180–194. doi:10.1145/3477132.3483591

[36] Kun Ren, Jose M Faleiro, and Daniel J Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1583–1598. doi:10.1145/2882903.2882958

[37] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. SLOG: serializable, low-latency, geo-replicated transactions. *Proceedings VLDB Endowment* 12, 11 (July 2019), 1747–1761. doi:10.14778/3342263.3342647

[38] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2012. Lightweight locking for main memory database systems. *Proceedings VLDB Endowment* 6, 2 (Dec. 2012), 145–156. doi:10.14778/2535568.2448947

[39] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2014. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings VLDB Endowment* 7, 10 (June 2014), 821–832. doi:10.14778/2732951.2732955

[40] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. 1976. The design and implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (Sept. 1976), 189–222. doi:10.1145/320473.320476

[41] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). *VLDB J.* (2007). doi:10.5555/1325851.1325981

[42] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 4–18. doi:10.1145/3514221.3526120

[43] Alexander Thomson and Daniel J Abadi. 2010. The case for determinism in database systems. *Proceedings VLDB Endowment* 3, 1-2 (Sept. 2010), 70–80. doi:10.14778/1920841.1920855

[44] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2213836.2213838

[45] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2014. Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems. *ACM Trans. Database Syst.* 39, 2 (May 2014), 1–39. doi:10.1145/2556685

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

[47] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1643–1658. doi:10.1145/2882903.2882934

[48] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 5–20. doi:10.1145/3035918.3064037

[49] Wikipedia contributors. 2024. Separation of concerns. https://en.wikipedia.org/wiki/Separation_of_concerns. Accessed: –.

[50] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 231–243. doi:10.1145/3183713.3196912

[51] Linguan Yang, Xinan Yan, and Bernard Wong. 2022. Natto: Providing Distributed Transaction Prioritization for High-Contention Workloads. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 715–729. doi:10.1145/3514221.3526161

[52] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 511–526. doi:10.1145/3318464.3389724

Received October 2024; revised January 2025; accepted February 2025

Table 2. Annotation results in descending order of the number of transactions. [M: Models, OT: One-shot Transactions, IT: Interactive Transactions, SIT: Strictly Interactive Transactions, CP: Complex Predicates, AD: Attribute Dependency, CM: CDA Mismatch, NE: Non-equality, TO: TypeORM application]

Owner/Repository	Description	Stars	Githash	M	OT	IT	SIT	CP	AD	CM	NE	TO
getsentry/sentry	Monitoring	37712	669d6c3	260	2549	230	19	20	18	1046	93	
saleor/saleor	E-commerce	20385	e78902b	122	1688	36	0	8	3	740	43	
openedx/edx-platform	Online courses	7120	765f2d8	200	1294	80	1	11	8	435	46	
pretix/pretix	Ticket sales	1824	1b72581	98	1000	232	48	33	81	391	72	
zulip/zulip	Chat	21280	88e91fa	100	1089	123	18	15	30	370	44	
misskey-dev/misskey	Blogging	9715	ae21b75	65	1086	7	0	15	0	348	55	x
ever-co/ever-gauzy	Business management	2120	4ffd8ec	141	1008	1	0	11	0	259	47	x
PostHog/posthog	Analytics	19823	c41995f	101	852	26	0	10	2	346	30	
bram2w/baserow	Database tool	2322	af01ec7	155	678	184	12	2	127	185	13	
grafana/oncall	Incident management	3367	01cb87c	85	808	49	2	3	0	240	21	
shuup/shuup	E-commerce	2214	25f78cf	155	784	42	0	20	2	333	16	
medusajs/medusa	E-commerce	24024	cff54d7	82	551	233	0	17	17	167	1	x
GreaterWMS/GreaterWMS	Inventory management	3831	ec7db97	35	766	0	0	33	0	188	35	
yogeshojha/rengine	Security	7245	9b6ccb3	49	733	0	0	66	0	340	19	
jumpserver/jumpserver	Access management	24734	4ebcba8	150	640	23	0	34	1	233	16	
uselotus/lotus	Pricing and billing engine	1699	0ea71f7	50	648	0	0	6	0	159	36	
WeblateOrg/weblate	Localization	4397	768f476	133	581	59	9	5	3	227	34	
mozilla/kitsune	Support website	1276	2a939c5	71	609	5	0	20	0	275	60	
samuelclay/NewsBlur	Newsfeed	6787	56cd514	18	565	0	0	10	0	173	35	
ansible/awx	DevOps	13746	a15bcf1	104	523	25	0	2	2	254	26	
inventree/InvenTree	Inventory management	3960	6700a46	109	450	69	0	10	1	144	24	
MicroPyramid/Django-CRM	Customer relationship	1891	11184ce	31	517	0	0	26	0	252	24	
netbox-community/netbox	Network automation	15492	0dde0b5	160	480	22	0	8	0	208	9	
bookwyrm-social/bookwyrm	Social network	2187	c4b21ee	74	433	41	1	2	5	135	12	
Flagsmith/flagsmith	Configuration management	4628	c896c50	98	437	1	1	4	0	140	29	
mozilla/pontoon	Localization	1437	fadf697	39	388	31	1	5	9	202	20	
amidaware/tacticalrmm	Monitoring	2981	8cfba49	41	411	3	0	4	1	93	10	
rafalp/Misago	Forum	2511	abad4f0	48	374	40	0	6	2	127	30	
LibrePhotos/librephotos	Photo management	6718	d0365b0	13	392	0	0	2	0	121	20	
wger-project/wger	Fitness	2974	2b6c528	59	378	0	0	3	0	79	6	
reviewboard/reviewboard	Code review	1547	d0e6a04	34	341	2	0	1	0	139	9	
allegro/ralph	Data center management	2197	5440253	154	312	29	0	3	5	129	10	
modoboa/modoboa	Mail hosting	2982	07b8cf9	38	337	3	0	14	0	76	8	
wagtail/wagtail	Content management	17667	252bae9	66	284	35	0	0	0	124	8	
readthedocs/readthedocs.org	Documentation	7950	757984f	60	263	1	0	1	0	108	22	
n8n-io/n8n	Workflow automation	42978	ee582cc	20	234	16	0	9	3	55	7	x
translate/pootle	Localization	1487	1e3fc44	51	235	6	0	8	0	94	5	
django-oscar/django-oscar	E-commerce	6181	0247120	144	231	9	1	13	0	89	21	
zhoubear/open-paperless	Document management	2559	b42d4e3	75	224	10	0	1	0	46	4	
blackholll/loonflow	Workflow management	1904	b0e236b	19	234	0	0	8	0	52	4	
paperless-ngx/paperless-ngx	Document management	18332	3d56a56	32	218	5	1	12	0	81	2	
QingdaoU/OnlineJudge	Competitive programming	5960	c25a314	14	207	11	0	9	0	88	11	
photonixapp/photonix	Photo management	1822	8a02fb3	11	215	2	0	4	0	70	8	
django-cms/django-cms	Content management	10057	5ce5586	24	182	14	0	1	1	80	3	
netless-io/flat-server	Classroom	638	4776c28	20	172	24	1	0	0	50	1	x
sct/overseerr	Media management	3604	77a33cb	11	190	0	0	0	0	35	2	x
fantasticit/think	Knowledge management	1934	1c2d133	10	184	0	0	1	0	47	0	x
healthchecks/healthchecks	Monitoring	7687	46c70a6	21	171	5	0	0	1	39	16	
laudspeaker/laudspeaker	Customer relationship	2056	9da9482	16	149	20	3	2	9	29	9	x
certsocietegenerale/FIR	Incident management	1705	97fc077	38	164	0	0	2	0	70	5	
apitable/apitable	Project management	12731	404dedb	33	160	2	0	0	0	35	2	x
intelowlproject/IntelOwl	Security	3203	f13a0d3	37	161	0	0	4	0	47	5	
jly8866/archer	DevOps	1559	e208c19	17	155	3	1	24	1	29	20	
mljar/mercury	Notebook converter	3873	fc0d4b2	18	149	8	0	1	1	38	3	
Tencent/CodeAnalysis	Static code analysis	1626	386c948	29	153	0	0	0	0	64	3	
khoj-ai/khoj	AI assistant	12103	ac3e508	29	152	0	0	1	0	41	5	
mathesar-foundation/mathesar	Database tool	2295	5bbcbbc	19	134	0	0	0	0	41	0	
mozilla/fx-private-relay	Email address generator	1444	9d2ae6e	15	127	4	0	1	1	29	7	
CromwellCMS/Cromwell	Content management	683	ce836a3	31	130	0	0	4	0	30	3	x
anyant/rssant	Newsfeed	1566	a6e37fb	18	100	24	0	2	2	41	6	
fantasticit/wipi	Blogging	615	9e3c15b	12	124	0	0	7	0	28	0	x
PokeAPI/pokeapi	Hobby	4067	f464535	217	120	0	0	3	0	36	3	
babybuddy/babybuddy	Baby tracking	2020	ba46c49	20	120	0	0	0	0	16	11	
nitely/Spirit	Forum	1160	069e82a	17	102	6	0	0	0	39	4	
tellery/tellery	Monitoring	352	0f0e1d2	16	95	9	2	1	0	12	3	x
ever-co/ever-traduora	Translation management	1968	d1cd6bc	11	90	9	0	0	0	21	0	x
stephenmcd/mezzanine	Content management	4747	e719286	35	97	0	0	1	0	33	0	
sissbruecker/linkding	Bookmark management	5423	fa5f78c	9	96	0	0	1	0	35	0	
cool-admin-midway	Admin dashboard	2434	edc1383	25	93	0	0	0	0	20	2	x
birkir/prime	Content management	1718	336f50c	11	92	0	0	2	0	23	1	x

Table 3. Annotation results in descending order of the number of transactions (continued).

Owner/Repository	Description	Stars	Githash	M	OT	IT	SIT	CP	AD	CM	NE	TO
genaller/genal-chat	Chat	1900	3c3c3bb	6	85	0	0	1	0	16	0	x
emqx/MQTTX	Connection management	3697	f589c82	10	81	0	0	0	0	6	0	x
l3tnun/EPGStation	Media management	538	c0e201b	10	66	12	0	4	0	22	9	x
doccano/doccano	Machine learning annotation	9272	4ac18ed	40	74	2	0	0	0	21	3	
iam4x/bobarr	Media management	1476	0fdf960	10	59	17	9	0	9	9	0	x
gristlabs/grist-core	Online spreadsheet	6675	1152976	18	46	29	0	0	12	8	3	x
antoinejaussoin/retro-board	Collaboration	773	c22ada6	16	16	59	0	0	4	2	0	x
django-wiki/django-wiki	Knowledge management	1808	050f124	17	69	2	0	1	0	15	1	
mdn/kuma	Documentation	1930	ae08600	16	70	0	0	3	0	11	1	
Gerapy/Gerapy	Crawler management	3278	e5662e2	6	70	0	0	0	0	3	0	
bpatrik/pigallery2	Photo management	1711	2aea6f4	12	68	0	0	6	0	14	3	x
owid/owid-grapher	Data visualization	1356	c666571	13	44	21	5	0	4	5	0	x
awesto/django-shop	E-commerce	3187	13d9a77	28	60	4	0	0	0	16	4	
wenqiyun/nest-admin	Permission management	577	ca53149	11	41	23	0	2	0	15	1	x
brookshi/Hitchhiker	Testing	2193	318bc47	20	54	8	0	0	0	12	3	x
dj-stripe/dj-stripe	Payment	1589	fa3990f	56	55	6	4	0	0	12	1	
hedgedoc/hedgedoc	Collaboration	4904	7286589	15	59	1	0	0	0	7	0	x
django-cms/django-filer	File management	1732	0ae797c	9	59	0	0	0	0	28	1	
liangliangyy/DjangoBlog	Blogging	6471	32158b3	16	53	2	0	0	0	15	3	
arxiv-vanity/arxiv-vanity	Paper renderer	1599	f7eb2f1	4	50	0	0	1	0	20	4	
jazzband/django-silk	Profiling	4348	1cb4623	5	41	4	0	0	0	27	2	
ansible-community/ara	DevOps	1827	8eda9c8	12	44	0	0	7	0	18	1	
OpenDroneMap/WebODM	Image processing	2758	fa058e7	8	41	2	1	0	0	3	1	
Koed00/django-q	Distributed task queue	1805	85baacc	5	38	2	1	0	0	13	5	
milesmcc/shynet	Analytics	2870	c4410c4	6	31	0	0	0	0	11	7	
ArkEcosystem/core	Blockchain	336	341c1e5	4	23	7	0	0	1	8	4	x
graphite-project/graphite-web	Monitoring	5875	0ec7201	14	30	0	0	0	0	2	2	
okfn-brasil/serenata-de-amor	Government data	4512	e7aeba7	6	28	0	0	0	0	10	0	
overshard/timestrap	Time management	1683	a063302	9	27	0	0	0	0	3	2	
weseek/growi	Collaboration	1297	941c546	4	24	0	0	0	0	5	0	x
chrisclark/django-sql-explorer	Database tool	2695	a5d678b	6	23	0	0	0	0	6	1	
shidenggui/bloghub	Blogging	413	314e309	2	22	0	0	0	0	8	0	x
gradejs/gradejs	Security	407	ceca9c9	10	17	2	2	3	2	4	2	x
pinry/pinry	Image board	3033	fd116a5	7	18	0	0	0	0	3	0	
boxyhq/jackson	Authentication	1708	4177982	3	15	2	0	1	1	1	1	x
decompiler-explorer	Decompiler	2000	0df1700	5	13	0	0	0	0	4	2	
mwmbl/mwmbl	Search engine	1453	fd35442	5	13	0	0	0	0	2	0	
scrapinghub/portia	Web scraping	9231	606467d	2	12	0	0	0	0	0	0	
garden-io/garden	DevOps	3297	8ccd77f	4	10	1	0	0	0	0	0	x
nukeop/nuclear	Music streaming	11808	0da69f7	3	10	1	0	0	0	1	0	x
tubearchivist/tubearchivist	Media management	4476	0110736	2	3	0	0	0	0	0	0	