



Data Persistence

BROUGHT TO YOU IN PARTNERSHIP WITH





Key Research Findings

An Analysis of Results from DZone's 2021 Data Persistence Survey

By John Esposito, PhD, Technical Architect at 6st Technologies

In January 2021, DZone surveyed software developers, architects, and other IT professionals in order to understand how persistent data storage and retrieval pathways are designed and evaluated.

Major research targets were:

1. Distribution of data persistence logic from file system to application levels (i.e., what part of the supra-OS stack decides how to store and retrieve data).
2. Thought processes of software professionals regarding data persistence.

Methods:

We created a survey and distributed it to a global audience of software professionals. Question formats included multiple choice, free response, and ranking. Survey links were distributed via email to an opt-in subscriber list, pop-ups on DZone.com, and short articles soliciting survey responses posted in a web portal focusing on database-related topics. The survey opened on December 23rd, 2020 and closed on January 12th, 2021, with 981 total responses recorded.

In this report, we review some of our key research findings. Many secondary findings of interest are not included here. Additional findings will be published piecemeal on DZone.com.

Research Target One: Distribution of Data Persistence Logic

Most software systems implement models of the world. Since models are abstract and, therefore, informationally incomplete, any software representation of a domain lacks some information relative to that domain.

The top of the software stack — the application layer — models the specific physical object, business process, etc. of interest to the developer (the “application”); the bottom of the stack — hardware controllers — models the physical device that encodes Turing-susceptible bits (storage and instructions).

In-between layers, from interpreters and compilers to operating systems and OS-specific hardware drivers, translate from top to bottom via additional abstraction, often (not always) losing more information at each translation step. We wanted to understand how software professionals approach the trade-offs required by information-destroying “impedance mismatches” at each level.

Additionally, dedicated DBMSs are designed to handle the largest chunk of “hoist-up” and “transmit-down” work along this stack. But there are many more storage and retrieval algorithms than there are database engines or popular hardware specifications, on the one hand, and vastly more application-level models than DBMS models (relational, column-oriented, etc.). Yet application developers are likely to influence DBMS selection, not least because changes in top (move to/from DBMS from/to application code) access patterns are more frequent than changes in bottom (move to/from DBMS from/to physical storage).

We wanted to understand how application developers decide how to approach DBMS type selection — especially relational database design. Much real-world software models business processes that are sufficiently actuarial to be represented naturally as tables (think “International *Business* Machines” — not “International *Social Graph* Machines”).

This was especially true during the early age of operations and database theory research (1960s-70s) when university curricula

in what is now designated “computer science” and “software engineering” were first developed. But the rise of the personal microcomputer, and in parallel the explosion of computing power that permits richer modeling of natural systems (molecular, meteorological, social, etc.), have increased the proportion of real-world systems modeled by software that are not naturally modeled as tables with relational keys — or at least, *prima facie* don't *seem* to be.

This explosion in application-level modeling variability, and again in the speed of model invention and modification, has tended to shift the programmer’s mind away from the tables-and-keys mode, even while the relational model remains dominant among mature DBMSs and database-related coursework.

In object-oriented programming, the frustration caused by this widening gap is captured by the phrase “object-relational impedance mismatch”; but many other mismatches are possible and are observed in practice (e.g., the “functional-graph” mismatch). We wanted to understand how software professionals approach these impedance mismatch problems, specifically object-relational mismatches, and in retrospect, how satisfactory their approaches have proved.

PREVALENCE OF IMPEDANCE MISMATCHES: DOMAIN MODEL VS. READ/WRITE PERFORMANCE

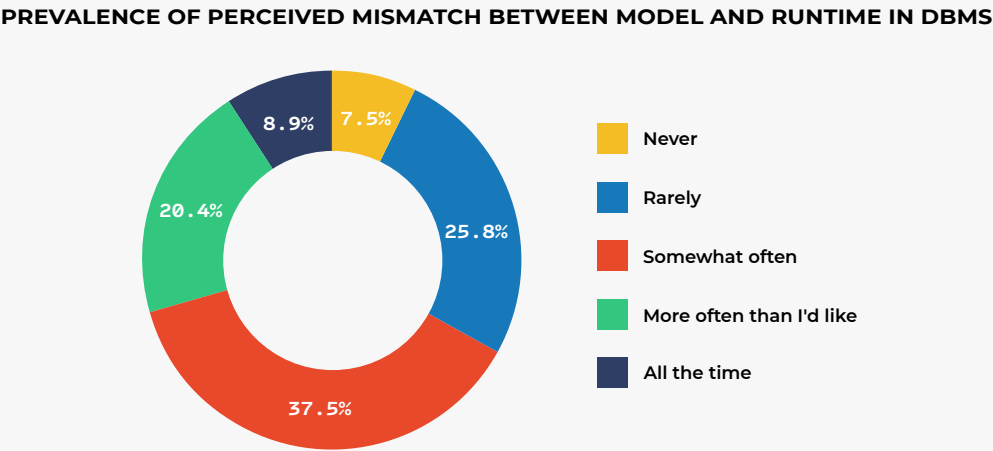
We wanted a high-level view of how often some mismatch between the domain model (addressed by the application layer) and the physical execution resulted in performance degradation.

We asked:

How often have you thought “the DBMS we’re using cannot simultaneously (a) effectively model this domain and (b) read/write performantly”? Consider any definition of “effectively” and “performantly.”

Results (n=953):

Figure 1



Observations:

- 1. Two thirds of respondents (66.8%, n=636) observed some mismatch between the DBMS’ ability to model the domain and its ability to read/write performantly.

This is a significant majority, but based on our own implementation experience and on countless anecdotes from other developers, the size of this majority is not surprising. Presumably, this means that DBMSs need to catch up with application domains — either DBMS technology itself, selection and tuning of the DBMS, or developer expertise with respect to the DBMS.
- 2. Technical role had some impact on response. Developers were more likely to respond “somewhat often” (38.9%, n=189) than technical architects (33.3%, n=38), but architects were slightly more likely to respond “more often than I'd like” (18.5%, n=90 among developers; 21.9%, n=25 among architects).
- 3. However, the line between developer and architect is often blurred, so we wanted to consider a more technical

segmentation. Since relational models remain dominant, we took “how often I write SQL” as a proxy for degree of in-the-weeds involvement with the relational DBMS. We then compared responses from those who write SQL every day vs. those who do not write SQL every day (for more details on this question, see discussion immediately below). Here, the differences were more pronounced but also more complex:

- Of those who write SQL every day, only 28.7% (n=60) reported a model/performance mismatch “somewhat often” vs. 39.9% (n=297) of respondents who do not write SQL every day.
- However, of those who write SQL every day, 17.2% (n=36) responded “all the time,” while only 6.6% (n=49) of those who do not write SQL every day responded “all the time.”
- We conjecture that those who write SQL every day are more likely to work on projects where an object-relational mapper (ORM) is not expected to operate cleanly, i.e., where some object-relational mismatch is known *a priori*. Therefore, more daily SQL writers encounter model/performance mismatches “all the time” because some model/performance mismatches are subtle enough to require less manual SQL writing.
- We suspect that the significantly higher “somewhat often” response from those who do not write SQL every day captures the prevalence of everyday model/performance slippage. That is, there are many, but not disastrously many, projects where a mismatch is not obvious *a priori*, or is obvious but does not affect performance egregiously enough to require manual SQL tuning.

HOW FREQUENTLY SQL IS WRITTEN MANUALLY

Although SQL syntax is simple, and most adults learned Venn diagrams in elementary school, writing clean and performant SQL, and designing databases to support queries required by the application (both data modeling and tuning performance [e.g., setting up indexes]), are not trivial. But the relational algebra implemented by SQL is conceptually somewhat far from both procedural and object-oriented programming paradigms (and perhaps less far from type-oriented functional paradigms), and simple SQL queries can be tedious to write.

Consequently, ORMs, SQL generators, and more specialized data access APIs (e.g., the DAO paradigm), designed and configured with the physical storage in mind, can sometimes produce better SQL than an application developer (whose code may not have anything to do with physical storage). Anecdotally, some developers complain about writing SQL, while others complain about the laziness of developers that do not write SQL.

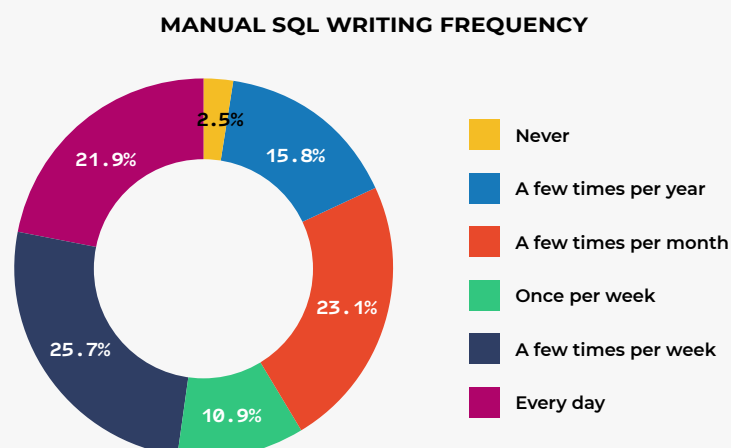
We wanted to know (1) how much developers think in relational models directly and (2) how well RDBMS automation works.

We asked:

How often do you write SQL manually?

Results (n=953):

Figure 2



Observations:

- 1. Nearly half (47.6%, n=454) of respondents write SQL more often than once per week. This is a larger percentage than we expected (although we did not have a specific percentage estimate). Presumably, this means that half of developers often move back and forth between relational and non-relational thought patterns.
- 2. After noting this high number, we hypothesized that responses might vary by company size, on the assumption that larger companies are more likely to use older technologies that rely more deeply on highly tuned RDBMSs and handle more actuarial (i.e., naturally tabular) data.

In fact, no linear correlation between SQL-writing frequency and company size was observed. Rather, large and small companies diverged at extremes of SQL-writing frequency.

Respondents at companies with 1,000+ employees were slightly more likely to write SQL every day (22.5%, n=79, vs. 20.6%, n=92 at companies <1,000 employees), but also slightly more likely to write SQL only a few times a year (18.5%, n=65, vs. 14.3%, n=64 at companies <1,000 employees).

- 3. Senior software professionals (<5 years of experience) are somewhat more likely to write SQL manually more frequently than junior professionals: While junior respondents were more likely to write SQL a few times per month (26.9%, n=56, vs. 20.8%, n=125 among senior respondents) or once per week (13%, n=27, vs. 10.8%, n=65 among senior respondents), senior respondents were more likely to write SQL a few times per week (27.8%, n=167, vs. 22.1%, n=46 for junior respondents) or every day (22%, n=132, vs. 19.2%, n=40 for junior respondents).

At a first glance, this may seem counterintuitive: Because SQL is relatively easy to learn, SQL writing might seem like a good task to assign to junior developers. The fact that this does not seem to happen, however, suggests (as anyone who spends much time writing SQL will probably eventually learn) that skill in writing SQL consists even less than usual in mastery of the syntax than in designing queries that match “above” (application domain) and “below” (table structure, storage model) effectively — the kind of trade-off-heavy work for which experience is especially helpful.

RELATIONAL DATABASE DESIGN

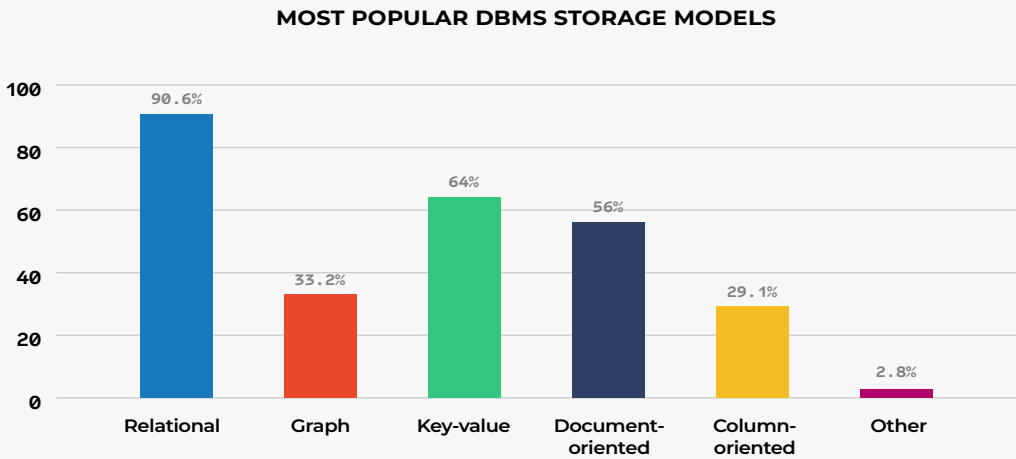
We wanted to know more about how software professionals model data relationally, because we assumed that RDBMSs remain the most popular DBMS paradigm by far.

We did empirically confirm this assumption, however, asking:

What type(s) of DBMS have you worked with?

Results (multiple selections were permitted):

Figure 3



Observations:

1. The overall rank of storage models is unsurprising. However, changes year over year are interesting, particularly in graph database usage. Time-series data on storage models, collected in this and other DZone surveys over the past several years, will be discussed elsewhere. For now, we note simply that our assumption regarding RDBMS' continued dominance is confirmed.

Given that relational databases are widely popular, we wanted to know more about how software professionals model data relationally.

HYPOTHESIS ONE

Hypothesis: The most popular normalization form is third normal form (3NF).

As a refresher, we define database normalization forms as follows:

- Non-normal: flat records, no uniqueness enforced
- First normal form (1NF): single value per column, uniqueness over all columns compounded
- Second normal form (2NF): 1NF plus single-column primary key
- Third normal form (3NF): 2NF plus no dependent non-primary key columns
- Boyce-Codd normal form (3.5NF): 3NF plus only one candidate key

(See [this article](#) for a brief review of normalization in relational database theory.)

Reasoning: (1) 3NF was the earliest relational schema defined as “perfectly” normalized ([Codd 1971](#)); and (2) anecdotally, 3NF most frequently appears in our own experience implementing relational databases, working with relational databases encountered in widespread consulting work, and attending software development conferences.

We asked:

Please rank how frequently you have enforced the following approaches to normalizing relations in a database (top=most frequently, bottom=least frequently).

Results (across all respondents; n=758):

Table 1

FREQUENCY OF ENFORCING DATABASE NORMALIZATION FORMS	
Normalization Form	Score
Second normal form (2NF)	3,506
First normal form (1NF)	3,184
Third normal form (3NF)	3,171
Non-normal	2,252
Boyce-Codd normal form (3.5NF)	1,987
Fourth normal form (4NF) or higher	1,345

Observations:

1. At first glance, our hypothesis appears to have been falsified. In fact, both second normal form and first normal form were used more frequently than third normal form. However, we believe this conclusion is misleading for the following reasons:
 - 1NF is not ranked significantly higher than 3NF.
 - Boyce-Codd normal form (sometimes called 3.5NF) was originally presented simply as an improvement of 3NF, not as

another type of normalization. Hence, 3.5NF is not always considered separately from 3NF.

- Because our question requested ranked ordering rather than absolute usage breakdown, combining results is a little iffy. However, if we combine scores (which are simply a sum of responses weighted by rank), then 3NF+3.5NF easily outscores other normalization forms (5158 weighted sum vs. next-highest 3506 for 3NF).

2. From these latter two points, we conclude that the hypothesis was, in fact, strongly verified, under the following modification: 3NF, *including 3.5NF*, is the most popular approach to relational data normalization.

HYPOTHESIS TWO

Hypothesis: Under-normalization is more likely to cause regret than over-normalization.

Models become more accurate in their domain, and domains themselves change over time. This fluidity becomes especially onerous when it affects data storage format: While changes to the model represented by application code generate new in-memory data structures at runtime, changes to the model represented by a database schema require offline changes of physical bits. Further, a database schema is an information bottleneck, so any changes to the bottleneck have far-reaching system implications.

We hypothesized that, in practice, people tend to normalize too little, and that they realize this only after initial schema design.

Reasoning:

1. Normalized schemas store data efficiently (without duplication) and group information clearly. If denormalization is needed for performance, it is relatively easy to flatten a normal schema. On the other hand, denormalized schemas often store data less efficiently (due to duplication) and can be harder to renormalize for technical reasons.

For example, it may not be clear what data is actually duplicate vs. identical; or the heap space required to hold normalized tables open may be large; or the number of database accesses required to translate from a flat table that exceeds available heap space may be large.

2. Further, laziness and deadlines exert some pressure toward under-normalization. More tables mean more commitment to a specific model, which takes more time to produce, and non-committal under-normalization seems immediately more attractive than incorrect normalization.

To test this hypothesis, we asked (with the same answer options as above [non-normalized...4NF]):

Please rank how frequently you have regretted enforcing the following approaches to normalizing relations in a database (top=most frequently, bottom=least frequently).

And:

Please rank how frequently you have regretted NOT enforcing the following approaches to normalizing relations in a database (top=most frequently, bottom=least frequently).

Results:

Table 2

FREQUENCY OF REGRET TOWARD NORMALIZING RELATIONS IN A DATABASE	
Regretted Enforcing	Regretted NOT Enforcing
3NF + 3.5NF	3NF + 3.5NF
1NF	2NF
Non-normal	1NF
2NF	Non-normal
4NF or higher	4NF or higher

Observations:

1. Bucketing 3NF and 3.5NF together, as we did earlier, results in the most-regretted enforcement or non-enforcement. This is simply because 3NF+3.5NF is far more common than all other types, and, therefore, reveals little or nothing about schematization regrets.
2. More informative are the rankings below top (and above 4NF), which confirm our hypothesis. That is, in the “fat part” of the bell curve, respondents were more likely to regret enforcing less normalization more frequently than they were likely to regret NOT enforcing more normalization.
3. Senior respondents (>5 years of professional experience in IT) are more likely to regret under-normalization than junior respondents. Among senior respondents, part from 3NF+3.5NF, non-normal is the most regretted approach, while among junior respondents, 1NF is the most regretted.

NOTE: It is hard to tell whether this difference is a function of different applications worked on, greater skill developed over time, or something else. For instance, one might imagine that, on the whole, Ajax-heavy modern web applications with high, small-request volume do better with document-oriented databases than with relational databases, and that older applications with lower request volume but more complex business logic might do better with relational databases.

If a larger portion of less experienced developers' careers has been spent on applications of the former type, then perhaps for them the normalization-needed curve is shifted toward “less normalization needed” by application-level requirements.

On the other hand, one might also imagine that more senior respondents are less likely to follow NoSQL trends blindly, simply because they are more accustomed to an RDBMS. Or again, perhaps the relational model, in reality, better suits many use cases that NoSQL databases are currently used for, and people with more experience are simply better at recognizing this. Further research is needed to tease out reasons for under- vs. over-normalization regrets.

ATTITUDES TOWARD OBJECT-RELATIONAL MAPPERS (ORMs)

The most common “impedance mismatch” is, presumably, between objects and tables. Where programs are written in an object-oriented paradigm, developers' minds are in object-space, not relation-space. In principle, object-relational mappers (ORMs) allow developers to treat tabular data as objects. In practice, the difference between object and relation can lead to performance degradation, debugging confusion, and hard-to-read code (where shims are developed to keep using the ORM but get around the mismatch).

We wanted to know how much trouble ORM really cause. To learn this, we again thought that developers' hindsight might reveal something about ORM success or failure. We asked:

Over the course of my professional career, I have used object-relational mappers (ORMs)...

- *More often than I should*
- *Less often than I should*
- *Just the right amount*
- *No opinion*

Besides retrospective evaluation, we also wanted to know whether developers use ORM more or less over time. (Perhaps developers keep wishing they had used ORM less but, in practice, end up using ORM no less, say, because imperfect design is less important than readable code.) We also asked:

Over the course of my professional career, I have used ORMs...

- *More frequently now than in the past*
- *Less frequently now than in the past*
- *The same amount now as in the past*
- *I have no idea*

HYPOTHESIS THREE

Hypothesis: Judged retrospectively, ORM's are used more often than they should be.

Reasoning: ORM's, in principle, produce (a) cleaner application code and (b) lower developer cognitive burden (by avoiding thought-pattern switches from object-oriented to relation-oriented). Both of these considerations appear more important prospectively and while coding than retrospectively. Further, real-world performance (as distinguished from pure algorithmic complexity) is often hard to judge prospectively, so performance losses caused by ORM's are more likely to appear in retrospect.

Results (n=932, for both questions):

Figure 4

ORM USAGE IN HINDSIGHT: PERCEPTION OF CORRECT USAGE FREQUENCY

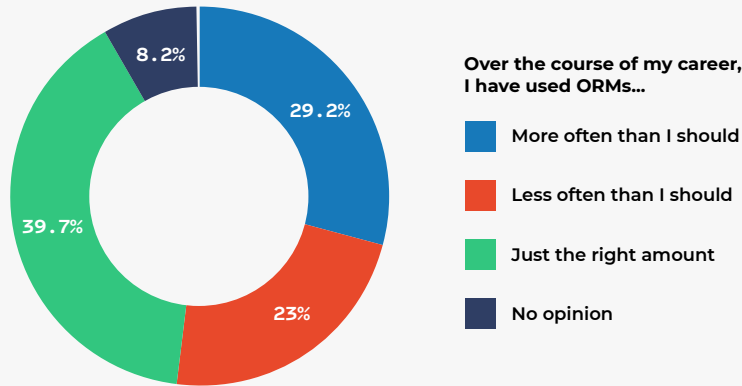
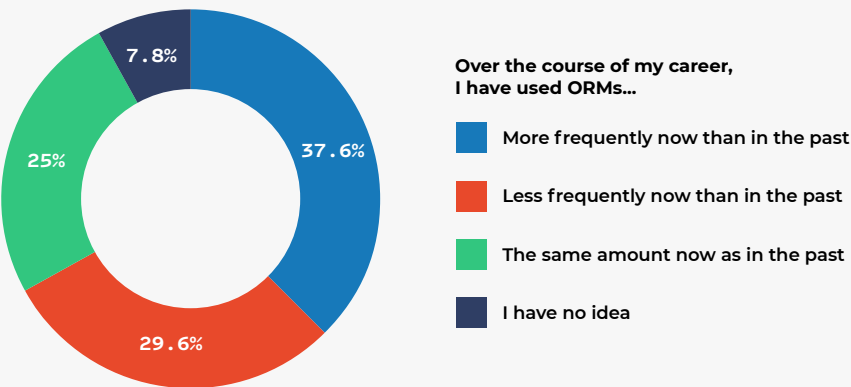


Figure 5

ORM USAGE IN HINDSIGHT: USAGE CHANGE OVER TIME



Observations:

- 1. A significant majority of respondents (52.2%, n=486) report that they do not use ORM's the right amount. This confirms that ORM technology and database design have not overcome the object-relational mismatch. However, the large minority of respondents that, in retrospect, judge they have used ORM's just the right amount (39.7%, n=370) suggests that the object-relational mismatch is far from catastrophic.

It would be interesting to compare retrospective judgments on other software design decisions. Based on our own experience, "we got it right 40% of the time" is actually an extremely impressive self-judged success rate in matters of software design.

- 2. Respondents are more likely to judge that they have used ORM's more often than they should (29.2%, n=272) than less

often than they should (23%, n=214). This confirms our hypothesis. We will continue to ask this question in future surveys. But we suspect that “used too much” will always exceed “used too little” given the necessary, intrinsic reasoning for our hypothesis — at least until formal software verification techniques encompass I/O details as well as control flow.

3. Despite their overall judgment that past ORM use has been excessive, significantly more respondents reported that they use ORMs more frequently now than in the past (37.6%, n=350) vs. less frequently now than in the past (29.6%, n=276). Note that the gap between “more” and “less” is greater in the case of the “action over career” question than in the case of the “retrospective judgment” question.

In other words, it seems not to be the case that retrospective judgment pressures against continued ORM usage. Perhaps this is partly because the intrinsic reasoning behind our hypothesis remains unchanged over the course of a developer’s career; perhaps ORMs have improved over time.

We are not aware of truly groundbreaking improvements in ORM technology over the past ten years. If you know any, and think that these improvements might help explain why developers use ORMs more frequently now than in the past, then please contact the [DZone Publications](#) team.

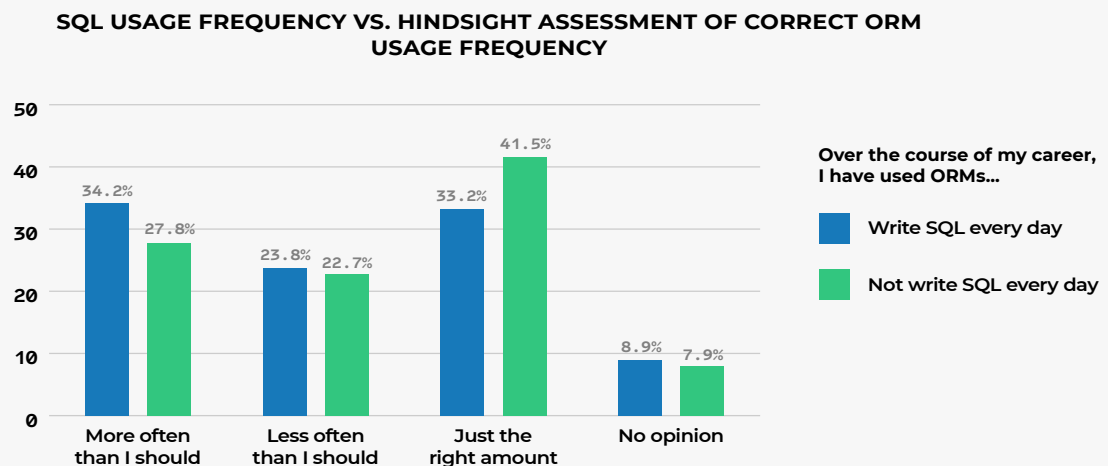
HYPOTHESIS FOUR

Hypothesis: People who write SQL every day are more likely to regret using an ORM.

Reasoning: People who write SQL every day are more likely to sense the gap between optimal queries, which they write daily, and queries resulting from ORM use vs. people who don’t directly optimize queries every day.

Results (n=932):

Figure 6



Observations:

1. The hypothesis was confirmed:
 - Respondents who write SQL every day are significantly more likely to report that they use ORMs more than they should (34.2%, n=69) vs. respondents who do NOT write SQL every day (27.8%, n=203).
 - Respondents who do NOT write SQL every day are more likely to report that they use ORMs just the right amount (41.5%, n=303) vs. those who write SQL every day (33.2%, n=67).

Research Target Two: Thought Processes of Software Professionals Regarding Data Persistence

DATA INTEGRITY IN SYSTEMS WITH POLYGLOT PERSISTENCE

Besides storage efficiency, the relational paradigm offers the ability to bake data integrity enforcement into the schema. For example, a relational schema that stores contacts and mailing addresses in separate tables, joined by an intermediate table

that defines a many-to-many relationship between contact and mailing address, ensures that the same physical mailing address is not represented inconsistently within the database. By contrast, a document-oriented database that stores contact name and mailing address as fields in a single flat document might easily represent the same physical address in slightly different representations.

For reasons of this kind, software systems are often built with specialized DBMSs underlying different applications but representing the same information. For example, a relational database that represents purchases might store each purchase as a row in a “transactions” table related to the purchaser, which is stored in a “buyers” table; but for time-series analysis of purchasing trends over time, where only the purchase datetime matters (and the individual purchaser does not), the same data might be loaded into a column-oriented database that stores purchases physically closer to one another based on purchase date.

In such “polyglot” systems, relational schemas can be used to place a DBMS-level constraint on data integrity across the system. This is an architecture-level way to have your RDMBS data pristineness cake and eat NoSQL performance too.

Since architectural decisions almost always involve trade-offs, the theoretical benefit of a relational “checkpoint” does not imply much about how the concept affects system design in practice. We wanted to know what software professionals actually think of the “relational checkpoint” concept.

We asked:

Agree/disagree: Every polyglot persistence system should include at least one relational “checkpoint” to enforce referential integrity at some specified level, time interval, or triggering event. By “polyglot persistence system,” we mean any system that includes both SQL and NoSQL DBMSs.

HYPOTHESIS ONE

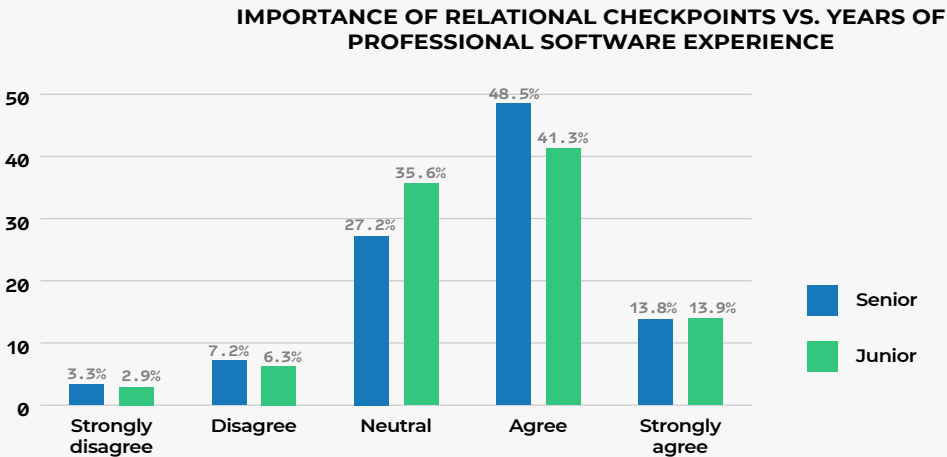
Hypothesis: More experienced software professionals are more likely to want at least one “relational checkpoint.”

Reasoning: Our reasoning for this hypothesis is allied to the reasoning for the hypothesis regarding under- vs. over-normalization regrets by experience level. The temptation to enforce data integrity in application code is stronger in inverse proportion to the amount of suffering caused by mistakes in application code, which is directly proportional to years of experience.

Another way of putting this is: Older developers are rationally lazier (in the sense of: are better at avoiding unnecessary grunt-work), so as they get older, they want the DBMS to handle more. (Yes, these are crude stereotypes. In fact, our guess is that the “mistakes in application code vs. years of experience” curve is logistic. We have no data on that beyond our own greying hair and memory of hotshotting yesterdays.)

Results (n=808):

Figure 7



Observations:

1. The hypothesis was confirmed, with a small complication. Senior (<5 years of experience as IT professionals) respondents were significantly more likely to agree with the “relational checkpoint” approach (48.5%, n=291, vs. 41.3%, n=86 for junior respondents).
- However, senior respondents were also slightly more likely to *disagree* with this approach (7.2%, n=43, vs. 6.3%, n=13 for junior respondents), while junior respondents were significantly more likely to remain neutral (35.6%, n=74, vs. 27.2%, n=163 for senior respondents).
- So seniority does tend to correlate with preference for a referential checkpoint in a polyglot system. But senior software professionals are also more opinionated about this design concept in both positive and negative directions.

CONSISTENCY VS. AVAILABILITY VS. PARTITION-TOLERANCE

Brewer’s theorem famously notes that a distributed datastore cannot simultaneously guarantee consistency, availability, and partition-tolerance. Since some CAP trade-off must be made, we wanted to know how software professionals think about this trade-off.

We asked:

Rank how much you’ve thought about the following guarantees of a distributed datastore over the course of your professional career.

And:

Rank how much you’ve thought about the following guarantees of a distributed datastore on recent projects.

HYPOTHESIS TWO (A)

Hypothesis: Software professionals now think more about availability than they did in the past.

Reasoning: Distributed datastores increasingly approach ubiquity as more systems run over the Internet, especially in the cloud, so more datastores are susceptible to availability issues.

HYPOTHESIS TWO (B)

Hypothesis: Partition tolerance remains the lowest concern but increases over time.

Reasoning: The Internet does an excellent job of recovering from partitions at the TCP level, but packet losses occur and out-of-order packet reconstruction (via packet sequence numbers) may come too late for data integrity (i.e., databases may need to support transactions that are non-commuting).

Results (n=901; score is sum weighted by rank):

Table 3

RANKING OF ATTITUDES TOWARD DISTRIBUTED DATASTORES	
Rank Over Career	Rank on Recent Projects
Consistency (score: 2137)	Availability (score: 2088)
Availability (score: 2071)	Consistency (score: 2021)
Partition-tolerance (score: 1211)	Partition-tolerance (score: 1269)


Observations:

1. Both hypotheses were verified:
- Over the course of their careers, respondents thought most about consistency. But on recent projects, respondents thought most about availability.

- Partition-tolerance remained by far the least important object of thought but marginally increased between whole-career and recent work.
2. The difference in weighted amount of thought reported between consistency and availability in both time periods is much smaller than the difference between partition-tolerance and the second-scored choice.

Future Research

This article is aimed mostly at application developers, and hence focuses on the interface between application code and the DBMS layer. Our survey included additional questions about interfaces between DBMS and lower layers (operating system, file system, physical storage device), including isomorphism between object models and physical data models; types of file systems personally implemented by respondents; how often respondents think about physical characteristics of secondary storage; how prospective consideration of physical storage details might impact performance; and more.

We are also collating data about storage models (relational, document-oriented, graph, etc.) usage over time and performance in theory vs. in practice with several years of data from previous DZone surveys. This and the lower-level information, aimed at DBMS designers, will be published in future articles on [DZone.com](https://dzone.com). 



John Esposito, PhD, Technical Architect at 6st Technologies

[@subwayprophet](#) on GitHub | [@johnesposito](#) on DZone

John Esposito works as technical architect at 6st Technologies, teaches undergrads whenever they will listen, and moonlights as research analyst at DZone.com. He wrote his first C in junior high and is finally starting to understand JavaScript NaN%. When he isn't annoyed at code written by his past self, John hangs out with his wife and cats Gilgamesh and Behemoth, who look and act like their names.



SingleStore

A Unified Database For Fast Analytics

More speed, scale, and agility in one powerfully simple,
cloud-native, distributed SQL database.

Try SingleStore Free

singlestore.com | [@singlestoreDB](https://twitter.com/singlestoreDB)

Case Study: Nucleus Security

Nucleus Security & SingleStore Partner to Manage Vulnerabilities at Scale

Nucleus Security was launched in 2018 by its three founders, Steve Carter, Nick Fleming, and Scott Kuffer, who set out to optimize VM processes and workflows by building a platform for the federal government, automating processes that had traditionally been done manually. Adhering to the federal government technology requirements, they chose to build the platform on MariaDB, as it was on the government's approved software list. Nucleus Security was able to ingest and process results from vulnerability scanning tools without issues when scanning was occurring monthly or quarterly. Yet when they uncovered opportunities to expand Nucleus Security to the private sector, many of their initial customers were scanning weekly or daily, and MariaDB became a bottleneck that was preventing the company from scaling to meet the needs of very large enterprises.

CHALLENGE

When the Nucleus Security team began architecting its platform, which was built to be deployed anywhere, but primarily in AWS, they explored graph databases, document-oriented databases, and traditional relational databases. But, said Steve, "It quickly became apparent as we started to onboard bigger customers that a high-performance relational database system was the correct solution to support our data model and the features we wanted to bring to the large enterprise market."

Scott added, "We were working on customer use cases that made it evident that we needed a relational database because all the data Nucleus is ingesting is highly relational. The majority of the data we store are time stamped vulnerability scan results, which include individual vulnerabilities, and each is tied to assets in a many-to-many relationship."

The team explored solutions for scaling MariaDB, like Percona, but found that it wouldn't meet their need to scale horizontally. "We determined that a sharded cluster was needed to scale the system horizontally. While Percona offered a clustered architecture it did not offer the performance improvements that a sharded system provides," said Steve.

RESULTS

SingleStore supports the MariaDB MySQL syntax, which enabled Nucleus Security to replace its original database with few code or query changes. The team got the SingleStore database deployed, its data loaded, and had the Nucleus Security application working in about half a day.

From COO Scott Kuffer "We closed a partnership deal with the Australian Post Office, our first cornerstone large enterprise client, which launched a lot of our subsequent success. We wouldn't have been able to support them without SingleStore."



COMPANY

Nucleus Security

COMPANY SIZE

11-50 employees

INDUSTRY

Tech, Value Management

PRODUCTS USED

SingleStoreDB

PRIMARY OUTCOME

- Successful market expansion into the large enterprise
- Ability to ingest 60X more assets compared to its legacy database
- Performance increase of 20X for its slowest scans

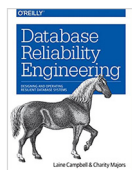
"Not only can we meet customer demands, but with SingleStore we've actually lowered our AWS infrastructure costs by about 3X."

— **Steve Carter**, Founder, Nucleus

Diving Deeper Into Data Persistence



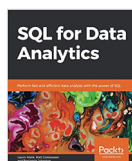
BOOKS



Database Reliability Engineering

By Laine Campbell and Charity Majors

This book explores core operational concepts that DBREs aim to master, examining how to implement key technologies to provide resilient, scalable, and performant data storage and retrieval.



SQL for Data Analytics

By Upom Malik, Matt Goldwasser, Benjamin Johnston

In this book, learn how to use SQL in everyday business scenarios efficiently and look at data with the critical eye of an analytics professional.



Python Data Persistence

By Malhar Lathkar

This book explores basic programming concepts, key concepts of OOP, serialization and data persistence in such a way that it is easy to understand.

REFCARDS

Getting Started With Distributed SQL

This Refcard serves as a reference to the key characteristics of distributed SQL databases and provides information on the benefits of these databases, as well as insights into query design and execution.

Hybrid Relational/JSON Data Modeling and Querying

For new applications, or those being refactored, now is the perfect time to adopt a hybrid relational/JSON data model to streamline development and provide greater flexibility in the future. Download this Refcard to learn more!

ZONES

Database The Database Zone is DZone's portal for following the news and trends of the database ecosystems, which include relational (SQL) and nonrelational (NoSQL) solutions such as MySQL, PostgreSQL, SQL Server, NuoDB, Neo4j, MongoDB, CouchDB, Cassandra, and many others.

Big Data The Big Data Zone is a prime resource and community for big data professionals of all types. We're on top of all the best tips and news for Hadoop, R, and data visualization technologies. Not only that, but we also give you advice from data science experts on how to understand and present that data.

TREND REPORTS

Data and Analytics

Getting to a place where a dashboard can be effective for all end-users, regardless of their understanding of data manipulation and analysis, is easier said than done. To help you avoid common pitfalls, make the most out of your data sets, and create robust visualizations, we've sought the opinion of experts in the industry to share their knowledge on what it takes to up your data analytics toolset.

The Database Evolution

How is the use of SQL and NoSQL databases evolving to support businesses' efforts to course correct in the big data era? In DZone's 2020 Trend Report "Database Evolution: SQL or NoSQL in the Age of Big Data," we explore the role of two popular database types SQL and NoSQL in big data initiatives over the next 6-12 months. Readers will find original research, interviews with industry experts, and additional resources with helpful tips, best practices, and more.

PODCASTS



DBAle

Each episode discusses topics and news in the world of SQL Server, while enjoying and chatting about beer. Cheers!



SQL Server Radio

From best practices for data storage to hierarchical partitioning — this podcast has you covered! Hosts Guy Glantser and Eitan Blumin talk all things SQL.



Microsoft Research, Databases

Check out Microsoft Research's Databases playlist on YouTube, containing 50+ talks and tutorials on databases.



INTRODUCING THE

Database Zone

Managing the growing amount of data effectively is an ongoing concern. From handling event and streaming data to finding the best use cases for NoSQL databases, there's plenty to unpack in the database community.

Keep a pulse on the industry with topics such as:

- Best practices in optimization
- Database performance and monitoring
- Handling event and streaming data
- Advancements in database technology

VISIT THE ZONE



TUTORIALS



CASE STUDIES



BEST PRACTICES



CODE SNIPPETS