DZone

THE DZONE GUIDE TO

DATA PERSISTENCE

VOLUME III

BROUGHT TO YOU IN PARTNERSHIP WITH

FairCom®

PERCONA

DEAR READER,

Say you're building an app. You're putting your data in boxes with labels like 0x9000000, messing with it a bit, maybe putting those boxes in other boxes. Sometimes you're working with the data, and sometimes you're holding the data so you can work with it later. Some things don't really fit in boxes though, so you start thinking in loops, then functions and monads (whether you call them that or not), then objects and messages (then maybe more functions). Now you can work with representations of users, handshakes, contracts, sensor readings, whatever. And then you think about users' names and locations, handshakes' protocols and signatures, APIs' methods and arguments... and then how all of these are related. You add new certificates and users, but not in order, so logically sequential data gets put in physically non-sequential boxes—and you have to start jumping over boxes. You decide to quit this box-juggling insanity and add new and powerful abstractions: first a nice file system, then a full-fledged database management system, and you're back to writing code.

Then maybe you start selling things and you add an Orders table. Now you have to rewrite half your code. So you hire a database administrator to handle that whole storage and query abstraction just so you don't have to worry about it anymore. Suddenly three tables with two inner joins and one (gasp) outer join swell to fifty, sixty, seventy billion rows, and it takes three hours to make a simple histogram for someone in the business strategy office, and you think do I really need anything except purchase sums for this graph? You read about column-oriented databases and get your DBA on board to spin up an analytics-only instance of Apache Cassandra. You've gone from x86 to Java to SQL to CQL and you're happy and hip – and you're all giddy to be writing actual code again.

That's how databases should work. They don't always. Maybe you and your DBA have conflicting constraints, or maybe you're your own DBA – and maybe you just want to keep things simple and clean (relational?), or maybe even just use the file system because you don't think you need the complexity and opacity of a DBMS. And is it really worth all that polyglot complexity just to store those user-user relationships in a specialized graph database?

Technical decisions around persistence are hard, but our 2016 Guide to Data Persistence will help you handle data better. We cover everything from current DBMS and ORM usage to modern database engines' data structures and access patterns to storing data on a mobile device. Give it a read and let us know what you think.

BY JOHN ESPOSITO
EDITOR-IN-CHIEF, DZONE

TABLE OF CONTENTS

EDITORIAL

John Esposito
research@dzone.com
EDITOR-IN-CHIEF

G. Ryan Spain
DIRECTOR OF PUBLICATIONS

Matt Werner
SENIOR EDITOR

Moe Long
EDITOR

Michael Tharrington
EDITOR

Tom Smith
RESEARCH ANALYST

Andre Powell
PROJECT COORDINATOR

BUSINESS

Rick Ross
CEO

Matt Schmidt
PRESIDENT & CTO

Jesse Davis
EVP & COO

Kellet Atkinson
VP OF MARKETING

Matt O'Brian
DIRECTOR OF BUSINESS DEVELOPMENT

Alex Crafts
sales@dzone.com
DIRECTOR OF MAJOR ACCOUNTS

Chelsea Bosworth
MARKETING ASSOCIATE

Caitlin Zucal
MARKETING ASSOCIATE

Aaron Love
MARKETING ASSOCIATE

Chris Smith
PRODUCTION ADVISOR

Jim Howard
SR ACCOUNT EXECUTIVE

Chris Brumfield
ACCOUNT MANAGER

ART

Ashley Slate
DESIGN DIRECTOR

SPECIAL THANKS to our topic experts, Zone Leaders, trusted DZone Most Valuable Bloggers, and dedicated users for all their help and feedback in making this report a great success.

WANT YOUR SOLUTION TO BE FEATURED IN COMING GUIDES?
Please contact research@dzone.com for submission information.

LIKE TO CONTRIBUTE CONTENT TO COMING GUIDES?
Please contact research@dzone.com for consideration.

INTERESTED IN BECOMING A DZONE RESEARCH PARTNER?
Please contact sales@dzone.com for information.

EXECUTIVE SUMMARY

For all the sophistication and reliability of the big three relational DBMSes, the ideal of optimal data storage and retrieval presents an ever-moving target. Data access patterns vary with main memory and SSD cost, client hardware capabilities, network reliability and throughput, user expectations, and application architecture—all of which are changing rapidly and often independently. DBMS technology is advancing at multiple levels, from new query languages to easier APIs to more diverse storage models, even to logical and physical structures implemented in innovative storage engines and local or distributed file systems. To help you navigate the sea of data persistence, we've focused this publication on three axes: how developers and DBAs are choosing the right DBMS (including how many they are using); which tools and abstractions developers are using to make data storage and retrieval easier to code and more robust under suboptimal runtime conditions; and what new database technologies are emerging. **This guide includes:**

- Expert knowledge for implementing data persistence and database best practices.
- A directory of tools to consider for storing and retrieving data.
- A checklist for choosing the right database for your use case.
- Analysis of trends in the space based on feedback from almost 600 IT professionals.

THE RELATIONAL PARADIGM REMAINS DOMINANT IN PRODUCTION, LESS SO IN NON-PROD ENVIRONMENTS

DATA In production environments, the two most mature commercial RDBMS offerings (Oracle and MySQL) are each used by about half of our respondents—by far the most popular DBMSes. Including SQL Server (at 34%), the top three DBMSes have remained well ahead of all competitors among both enterprise developers (represented by our survey respondents) and IT professionals at large (represented by the DBMS ranking aggregator db-engines.com). MongoDB comes in a distant fourth (21%), well ahead of other NoSQL data stores.

IMPLICATIONS Even as use cases multiply and applications grow more distributed, relational storage and SQL query access are not going anywhere—nor should they. The flexibility and low up-front commitment offered by document-oriented databases makes them an attractive second choice in both production and dev/prototyping environments, and the relative maturity of several document-oriented DBMSes (such as MongoDB and Couchbase) makes them suitable for production as well.

RECOMMENDATIONS Get sharper at SQL. Keep up with advances in RDBMS technology. (For more on some recent advances in storage engines see Vadim Tkachenko's article on page 8). Consider other storage models, but factor the additional decades of optimization enjoyed by SQL query and storage engines into your choice of DBMS.

POLYGLOT PERSISTENCE IS NOW VERY COMMON, THOUGH NOT VERY POLY

DATA Nearly as many respondents (38%) typically use two storage models in their applications as those who use one storage model (40%).

IMPLICATIONS "NoSQL" is better understood as "Not Only SQL" as interest grows in matching the persistence mechanism to the structure of the data to be persisted—tabular or not, independent of scale.

RECOMMENDATIONS Don't limit yourself to one storage model; consider which data and which access patterns are best suited to which model, and be willing to persist data in multiple appropriate stores. Microservices and containers may help keep multi-store architectures clean.

DBAS ARE HANDLING SCALE EFFECTIVELY

DATA 22% of respondents didn't know whether their databases were partitioned or not.

IMPLICATIONS Nearly a quarter of developers can build applications without knowing how their database handles scale—in other words, without worrying about volume-related decisions their DBAs are making for them.

RECOMMENDATIONS Hire excellent DBAs who know how to distribute large volumes of data under flexible read/write conditions. Storage abstraction should be as tight as possible—which means both an app-matched storage model and an appropriate partitioning policy.

MATCH BETWEEN DATA STRUCTURE/ACCESS AND STORAGE MODEL SHOWS ROOM FOR IMPROVEMENT

DATA Only 22% of respondents use a specialized graph DBMS to store data that is naturally modeled as a graph. ORMs remain popular among a significant majority (58%) of respondents.

IMPLICATIONS In many cases objects and tables do map very well—the so-called object-relational impedance mismatch often does not apply. But specialized object stores (e.g. PostgreSQL) do exist and map storage and access more straightforwardly than an ORM. The mismatch between graphs and relational stores is more severe, and the low rate of graph DBMS adoption may indicate both less familiarity with methods used to store and query graphs as well as less ecosystem maturity (at both DBMS and connector/framework levels).

RECOMMENDATIONS Use ORMs with eyes wide open, and consider an object-relational database (such as PostgreSQL or Informix) as an alternative. Build graph processing skills (which—from a hierarchical, object-oriented perspective—may seem trickier to acquire than they really are), and consider a graph-first DBMS (such as Neo4j or OrientDB). (For more on how to choose the right DBMS for your use-case, see the checklist on page 18.)

KEY RESEARCH FINDINGS

583 IT Professionals responded to DZone's 2016 Data Persistence Survey; the demographics of this survey are as follows:

- 69% of these respondents use Java as their primary programming language at work.
- 68% develop primarily web applications.
- 66% have been IT professionals for over 10 years.
- 45% work at companies whose headquarters are located in Europe, 27% in the USA.
- 44% work at companies with more than 500 employees, 19% at companies with more than 10,000 employees.

ORACLE, MYSQL, AND SQL SERVER REMAIN HEAD-AND-SHOULDERS ABOVE THE REST; ORACLE AND MYSQL REMAIN NECK-AND-NECK

The two most mature commercial DBMS offerings (Oracle and MySQL) are used in production by 51% and 49% of respondents, respectively—significantly ahead of the third-ranked DBMS (SQL Server, at 34%). The top three, and the tight race between the top two, have not changed in years, among our survey respondents as well as on the DBMS ranking aggregator db-engines.com. The nearest NoSQL challenger, MongoDB, remains a distant fourth in production environments.

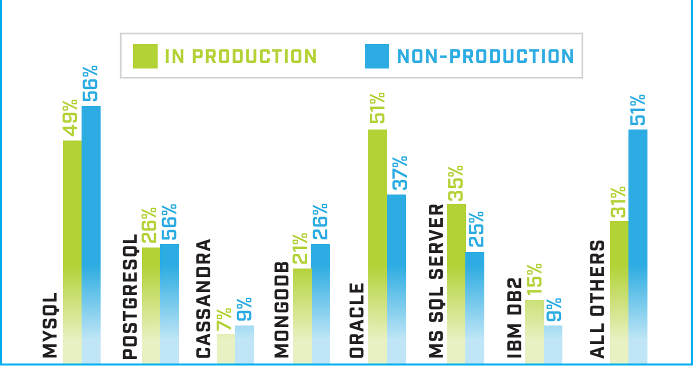
NOSQL—ESPECIALLY DOCUMENT-ORIENTED—DBMS ADOPTION IS SIGNIFICANTLY GREATER IN NON-PRODUCTION ENVIRONMENTS

Non-production environments are more friendly to less mature and less thoroughly supported database management systems and also more likely to be affected by desire to optimize for structural fit and ease of access. In production, where data stores are often managed by specialist non-developers, factors other than developer experience and optimal match between data processing and storage and retrieval algorithms weigh into DBMS selection more heavily. NoSQL and generally less mature and/or less supported offerings should therefore be more popular in non-production environments. DBMSes that implement simpler storage models well-suited to lightweight prototyping—especially, therefore, document-oriented DBMSes—should gain an extra boost in development environments.

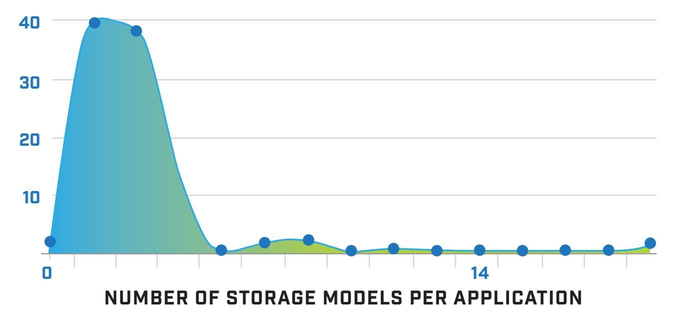
Accordingly, the gap between production and non-production usage is greatest for the two most mature commercial DBMS offerings: Oracle (at 51% in production vs. 37% in non-production) and SQL Server (34% in production vs. 25% in non-production), and the gap between the most popular NoSQL offering in production (MongoDB, at 20% adoption) and the least popular of the top three (SQL Server, at 34%) enters within the survey's margin of error in non-production environments (where MongoDB enjoys 25.4% adoption vs. SQL Server's 24.6%).

MongoDB's (static-schema-free) document orientation, familiar JSON-like document format (ordered lists supporting a variety of types), and widespread connector availability make it easy to set up without heavyweight data modeling and relatively straightforward to use for many less-data-intensive applications without cramping application architecture or code. Indeed, many non-relational stores are easier to spin up quickly than a full-power RDBMS. Some benefits of the relational model (especially integrity enforcement) are less relevant in non-production environments, where updates don't always need

01. **DATABASE MANAGEMENT SYSTEMS IN PRODUCTION VS. NON-PRODUCTION ENVIRONMENTS**



02. **POLYGLOT PERSISTENCE: HOW MANY PERSISTENT STORAGE MODELS DO YOUR APPLICATIONS TYPICALLY USE?**



to propagate across all entities. For a conceptual overview of key-value, column-oriented, document-oriented, and graph databases, see this recent article by Saravanan Subramanian. To map use cases to storage models, see the selection matrix on page 18.

Note: of the top three DBMSes, only MySQL enjoys greater adoption in non-production vs. production environments. MySQL is especially likely to be many developers' default non-production RDBMS, presumably because it is popular, open-source, mature, familiar, and supported by a strong community. (For the importance of familiarity in developers' preference for a particular data persistence technology, see the upcoming section on matching storage model to data structure.)

APPLICATIONS ARE ALMOST AS LIKELY TO USE TWO STORAGE MODELS AS ONE

Developers may use more than one storage model in different applications with no reference to the work done by the application; variety by developer speaks more about the human than about the technology. But variety of storage models within a single application indicates "polyglot" persistence—that is, how many storage models are used to persist data where technical and business needs overlap. Among our respondents, nearly as many respondents typically use two storage models in their applications (38%) as use one (40%). This result confirms that "NoSQL" is better understood as "Not Only SQL" because the most popular storage model (given DBMS and query language usage data) remains relational. Based on DBMS adoption data, the second most popular storage model by user count is probably document-oriented; but because other storage models (especially column-oriented, graph, and key-value) are particularly well suited to analytical processing of many data rows, further research is required to discover storage model usage by data volume. In any case, the near-parity between one and two storage models per application indicates increasing interest in matching persistence mechanism to the structures of data to be persisted. (For more on how to choose the right DBMS for your use-case, see the checklist on page 18.)

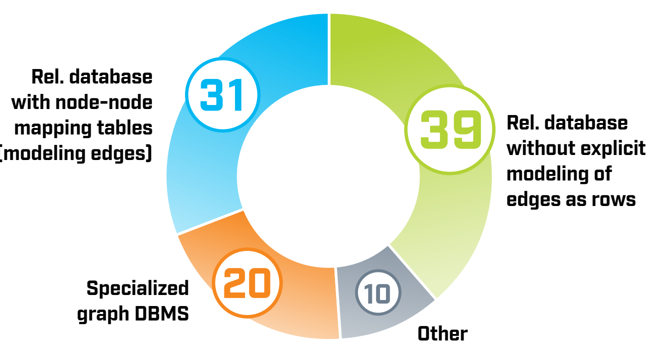
MATCHING STORAGE MODEL TO DATA STRUCTURE: MODELING GRAPH DATA

Graph structures do not fit the relational model comfortably. In a relational database, most (Shannon) information is stored in the columns and rows of each table; the schema is a technical construct designed to enforce data integrity, make the data model more legible, and make the querying model more efficient; not to encode more information. In a graph, however, most information is stored in the structure of the nodes and the edges; additional information about nodes and edges is treated as metadata. Yet many real-world entities are most naturally represented as graphs: social, travel, and trade networks; packet routes; control flows; etc. Storing graph structures in tabular storage is inelegant and inefficient even at first, static-only glance; but the problem gets worse in a dynamic setting. Because a graph's computational complexity may diverge wildly from its combinatorial complexity, reducing a graph to a relational schema (e.g. two-column mapping tables that relate a row in one table to a row in another—that is, modeling nodes as columns and edges as rows in a new table) may work far better for some algorithms than for others (in ways that are not immediately obvious from the graph itself).

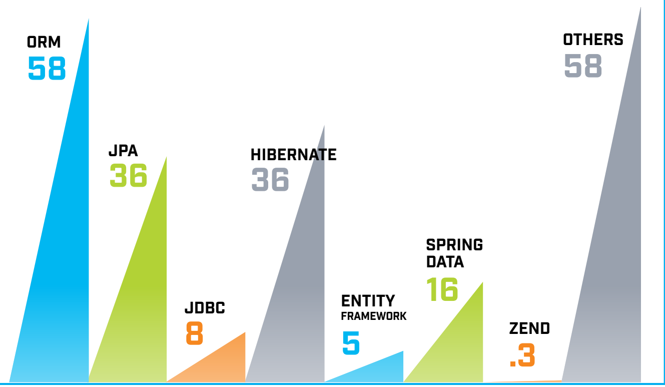
Nevertheless, three factors encourage developers and DBAs to store data that is naturally modeled as a graph in a relational DBMS: first, the maturity of relational DBMSes; second, the simplicity and familiarity of SQL (which 90% of respondents use regularly); and third, the availability and maturity of powerful object-relational mappers (ORMs) that make relational data easily accessible (often with automatic and highly effective optimizations) from application code.

Accordingly, only a small minority (20%) of respondents persist data that is naturally modeled as a graph in a specialized graph DBMS. Further, more respondents store naturally-graph data in a relational database without explicit modeling of edges as rows (39%) than with node-node mapping tables (31%). We expect this distribution to change as graph DBMSes and query languages grow more familiar, as tooling ecosystem around these DBMSes approaches the maturity of ORMs, as inefficiencies introduced by storage-structure mismatch grow more expensive as graph

03. **HOW DO YOU TYPICALLY PERSIST DATA THAT IS NATURALLY MODELED AS A GRAPH?**



04. **PERSISTENCE-RELATED TECHNOLOGIES RESPONDENTS MOST ENJOY WORKING WITH**



data volume increases, and as use cases (and corresponding storage and retrieval algorithms) grow more varied.

Two possibly linked correlations are also worth noting. First, the largest chunk of respondents who store graphs in a relational database without explicit modeling of edges use Oracle (25%)—probably the most mature and most thoroughly optimized RDBMS. Second, the largest chunk of respondents who store graphs in relational database WITH node-node mapping tables use MySQL (24%), which is also the only RDBMS that gains popularity in non-production vs. production environments. This difference may be a function of both the greater likelihood that MySQL will be used for experimental purposes – where graph problems, insofar as conceptually farther from actuarial use (for which relational databases are a more natural fit), are more likely to appear.

MATCHING PROCESSING APPROACH TO STORAGE MODEL: USE AND ENJOYMENT OF ORMS

Most developers use SQL (90%) but the relational algebra does not naturally capture object-orientation. Objects do not fall into Venn diagrams; but objects and relational tables do share enough structure that, for many simple (few-join) access patterns, the so-called object-relational impedance mismatch does not cause catastrophic performance or integrity loss. Accordingly, object-relational mappers (ORMs) are not only widely used, but also preferred by a majority of developers. In response to our question, "What persistence-related technology do you most enjoy working with?" 58% of respondents answered that they most enjoy working with ORMs. Of these, 70% specifically enjoyed working with Hibernate—probably a function of both Hibernate's maturity and also our respondents' heavy focus on Java. Although the tail of most-enjoyed data persistence technologies was quite long (26 distinct technologies), Spring Data emerged as the most popular comprehensive data access framework by far (16%).

REASONS DEVELOPERS ENJOY WORKING WITH A DATA PERSISTENCE TECHNOLOGY

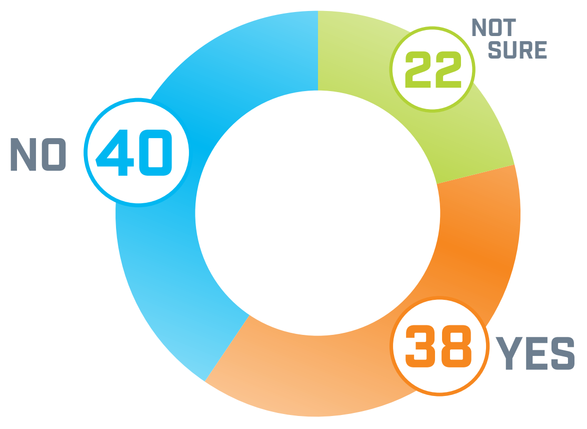
Just under two-thirds of all respondents who named the persistence-related technologies they enjoy working with also specified why they enjoyed working with those technologies. Grounded-theoretic "bucketing" analysis yielded seven (somewhat overlapping) reasons to enjoy a persistence technology (listed in order of popularity): ease of use, simplicity, adherence to standards, familiarity, performance, high level of control, and scalability. The most popular reason by far was ease of use (34%), followed by simplicity in distant second (21%). The top four reasons relate more directly to developer experience than to outcomes (such as performance and scalability), as the wording of the question ("enjoy") indicated. Additional research is required to determine how developer experience relates to persistence-related technology selection, especially because many less-familiar (NoSQL) technologies are optimized for scalability and general performance for certain use cases. (For more on how to choose the right DBMS for your use case, see the checklist on page 18.)

HANDLING SCALE: DATA IS PARTITIONED AS FREQUENTLY AS IT IS NOT, BUT THIS IS OFTEN SUCCESSFULLY MADE INVISIBLE TO DEVELOPERS

Modern storage engines, across all storage models, are highly optimized for current hardware, access patterns, and network performance. Theoretically massive inefficiencies of the relational storage model sometimes dominate the advantages offered by a higher degree of maturity among RDBMSes, although newer engines store data in structures that are less narrowly tuned to read-heavy loads using slow (spinning) physical media than (for example) B+ trees. But as Big Data strategies aggressively drive data storage and processing needs, data scale becomes increasingly difficult to manage.

To keep performance and availability high, data is often partitioned on physical and logical lines. Among our survey respondents, 38% partition data in some way (vertical, horizontal, or functional) vs. 40% who do not—a difference within the survey's margin of error (5%). Two research followups would prove interesting: first, what specific data volumes (or velocities), application requirements, and infrastructure constraints drive what kinds of partitioning; and second, which storage models are more likely to require partitioning (although application constraints presumably affect both choice of storage model and partition size/need). It would appear, however, that distributed data techniques designed to manage CAP trade-offs are often effective: 22% of respondents—most of whom are developers and not DBAs—were not even aware of whether or not their databases were partitioned—a sign that, for nearly a quarter of developers, physical splitting of data had no visible impact on their development work.

05. **DATABASE PARTITIONING: DO YOU SPLIT UP YOUR DATABASE WHEN IT GETS TOO BIG?**



Queries Tell the Story

Query performance is one way to look at MySQL performance. MySQL should react to application queries quickly, with the correct responses. Analyzing query response times is often a good way to spot problems.

From an application standpoint, query responses consist of three different parts: the MySQL server running the query, network contributions, and application processing.

There are a number of tools that let you examine server query patterns (Percona Toolkit, SolarWinds Database Performance Analyzer, MySQL Enterprise Monitor, etc.). These tools reveal which queries are being executed and their response times. Captured at regular intervals, this information can reveal application-side performance problems:

- If the MySQL server is getting the same query mix, and the response times are the same, it is unlikely that it is a MySQL server problem.
- More frequent specific queries might be an application-side problem. For example, a malfunctioning cache often has a major inflow of specific queries. An increase in traffic often causes a proportional increase in queries.
- Less inflow of some (or all) queries might also indicate an application-side problem. For example, you might find a caching layer is overloaded, reducing the inflow of queries to MySQL. It could also be a MySQL issue. Take a look at the Processlist in this case. Severe bottlenecks, such as a locked table, might pass fewer (or block all) queries.
- MySQL might be the issue if a similar or smaller number of queries are processed, but with higher response times.

In addition to examining the "query fingerprint," investigate the number of threads running as plotted by the monitoring system. If there are more threads running than CPU cores available, chances are you might be dealing with a bottleneck related to CPU saturation, disk IO or table/row-level lock contention. In this case MySQL needs to be checked out.

For more specific help with query tuning, see the Percona Data Performance Blog, or our ebook, Practical MySQL Performance Optimization.

WRITTEN BY PETER ZAITSEV

CEO AND CO-FOUNDER OF PERCONA

Percona Server BY PERCONA

PERCONA

Enhanced, fully compatible, open source, drop-in replacement database software that provides superior performance, scalability and value.

CATEGORY	NEW RELEASES	OPEN SOURCE?
DBMS	Typically one quarter after Oracle and MongoDB GA releases; regular feature and maintenance updates.	Yes

STRENGTHS

- MySQL, MongoDB versions
- All features and benefits of community edition
- Enterprise-ready, with free enterprise features and functionality
- Optional storage engines included
- High performance and availability for optimal cloud deployment
- Includes Authentication and Audit Plugins

CASE STUDY

Big Fish is a huge mobile game provider, who has distributed more than 2 billion top-grossing games to date. As a data-driven company, Big Fish's database infrastructure must be fast, reliable, and scalable. They've standardized on Percona because the technology has the required features and performance, and our support team is always there to quickly resolve challenging issues. Big Fish relies on a number of Percona software solutions to optimize its database infrastructure, including Percona Server. Percona Server provides greater visibility into the database infrastructure than MySQL Server, with access to user statistics and response time distribution that enables administrators to improve capacity planning. This helps to control costs by reducing the amount of hardware that needs to be purchased.

In addition, Percona services such as consulting and support help Big Fish handle evaluating fast storage, performing upgrades, performance and configuration audits, as well as day-to-day operations.

NOTABLE CUSTOMERS

- Yelp
- Alcatel-Lucent
- Singular
- Facebook
- Cisco
- BBC

BLOG percona.com/blog | **TWITTER** @Percona | **WEBSITE** percona.com

How Three Fundamental Data Structures Impact Storage & Retrieval

QUICK VIEW

01
Data performance powers application performance, which in turn powers business performance.

02
Data performance needs to be tuned and managed in order to be correctly optimized for applications.

03
The type of data structure inherent in your storage engine can help optimize database performance.

BY VADIM TKACHENKO

As our daily dependence on applications grows, our expectations for those applications also grow. We want applications to be always up, bug free, easy to use, secure, and high performance.

It's a simple relationship: data performance powers application performance, which in turn powers business performance. And just like there are a growing number of applications that process data, there are an equally growing number of ways to store the data. How you store and retrieve that data matters.

In order to get peak performance, it is important to understand the differences between the storage engines. Using one of these algorithms can affect the way your queries perform. This article will discuss data storage algorithms and why you should care about how they operate.

DATA STORAGE AND RETRIEVAL

First, let's talk about how we interact with data. There are two primary data actions: store it, and later retrieve it. Above that, we apply some structure to the data. There are mainly two ways of doing this:

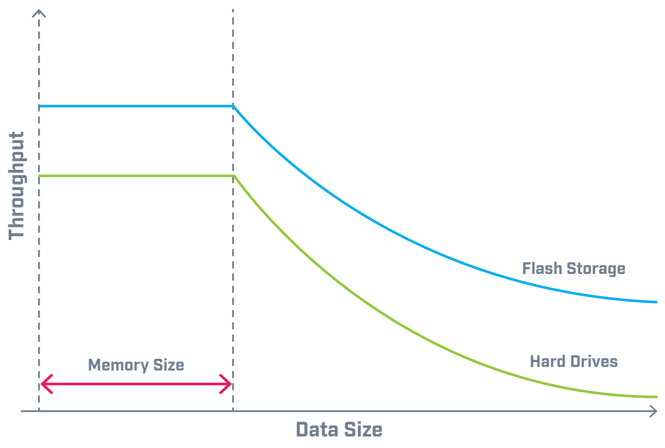
- A relational database management system (RDBMS) , or "SQL data."
- A non-relational database, or "NoSQL data."

While data can be stored in many different ways, we need to effectively organize the data in order to search it and access it. In the case of SQL and NoSQL, both solutions build special data structures called "indexes." The data structure chosen often helps to determine the performance characteristics of the store and retrieve commands.

B-TREES

A traditional and widely-used data structure is called "B-tree." B-tree structures are a standard part of computer science text books, and are used in most (if not all) RDBMS products.

B-tree data structures' performance characteristics are well understood. In general, all operations perform well when the data size fits into the available memory. (By memory, I mean the amount of RAM accessible by the RDBMS in either the physical server or virtual server.) This memory limit is usually a hard restriction. Below is a general chart I like to use to demonstrate B-tree performance characteristics.



This chart clearly illustrates a couple of points:

- As soon as the data size exceeds available memory, performance drops rapidly.
- Choosing a flash-based storage helps performance, but only to a certain extent—memory limits still cause performance to suffer.

B-tree-based structures were designed for optimal data retrieval performance, not data storage. This shortcoming created a need for data structures that provide better performance for data storage.

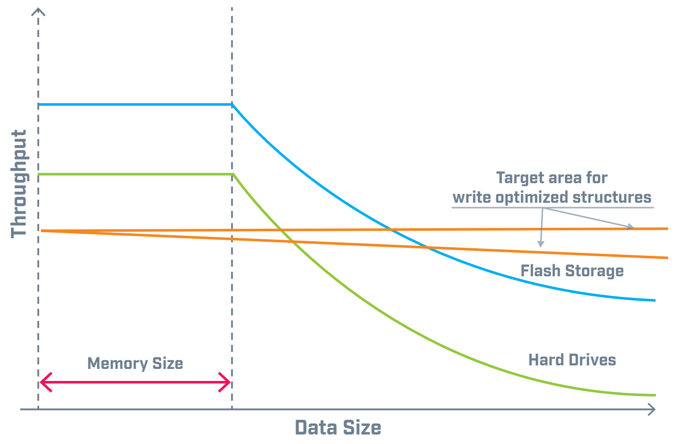
So when is B-tree a good solution for your applications? The chart above provides clues:

- When data size doesn't exceed memory limits
- When the application is mostly performing read (`SELECT`) operations
- When read performance is more important than write performance

Events that might exceed B-tree performance limits include: accepting and storing event logs; storing measurements from a high-frequency sensor; tracking user clicks; and so on.

In a majority of cases, it is possible to solve B-tree performance issues with more memory or faster physical storage (see previous chart). But when hardware adjustments aren't an option, a different data structure can help.

Two new data structures were created for write-intensive environments: log structured merge (LSM) trees and Fractal Trees®. These structures focus on data storage performance rather than data retrieval.



(Keep in mind that the graph shows asymptotic theoretical trends. Real performance graphs vary hugely with the specific implementation and software involved.)

Before going into LSM and Fractal Tree details, let's discuss the "key-value" concept.

In any storage structure, data can presented in a key => value format. This is familiar to NoSQL users, but probably not to RDBMS users. RDBMSes instead use primary keys associated with tables, and the data is internally represented as `primary_key => table_columns`.

For example, a user account may need the following data:

```
user_id; user_name; user_email; user_address;
account_name; user_zip; user_year_of_birth
```

This would be represented (using a primary key) as:

```
user_id => user_name; user_email; user_address;
account_name; user_zip; user_year_of_birth
```

Any of the following could be used as a secondary index:

- by email (for fast search by email): `(user_email => user_id)`
- by year of birth: `(user_year_of_birth => user_id)`
- by postal code: `(user_zip => user_id)`

Searching for a `user_name` by `user_email` would be done in two steps:

1. search `user_id` by `user_email`
2. search `user_name` by `user_id`

An "insert" operation to this table could look like this:

```
user_id: 1000; user_name: "Sherlock Holmes"; user_email:
"sherlock@holmes.guru"; user_address: "221B Baker Street";
account_name: "sherlockh"; user_zip: "NW1 6XE"; user_year_
of_birth: 1854
```

The transactions for this operation would be:

1. *Insert into primary data storage (or primary key):* `(1000 => "Sherlock Holmes"; "sherlock@holmes.guru"; "221B Baker Street"; "sherlockh"; "NW1 6XE", 1854)`
2. *Insert into email index:* `("sherlock@holmes.guru" => 1000)`
3. *Insert into year_of_birth index* `(1854 => 1000)`
4. *Insert into postal code index* `("NW1 6XE" => 1000)`

After the initial insert operation, there would be three additional operations behind the scenes (known as "index maintenance" overhead). This overhead can contribute to performance degradation.

Let's say we want to update an email record for a user (`user_id: 2000; new email: "newm@example.com"`). The following transactions would occur:

1. Primary data storage: find `user_id:2000`; read email to `old_email`; rewrite email to `"newm@example.com"`
2. Email index:
   A. Find record with `key = old_email`; delete it
   B. Insert record (`"newm@example.com" => 2000`)

Sequential keys (or monotonically increasing functions) generally don't cause problems for B-tree structures—it's random operations that cause performance hits. An email address is a good example of a random insertion.

So random operations make B-trees problematic, performance-wise, due to hardware limitations—random "modify" operations cause multiple disk IOs.

Both LSM and Fractal Trees attempt to improve performance by making key operations less random. These data structures also provide better compression and smaller write amplification, which is better for flash/solid-state storage.

LSM TREES

The first mention of LSM trees dates back to 1996, and corresponds to Google BigTables. Later it was implemented in products such as Cassandra, LevelDB, and most recently in RocksDB.

An LSM tree works by:

- Storing incoming modify operations in a buffer (usually named "memtable")
- Sorting and storing the data when the buffer is full

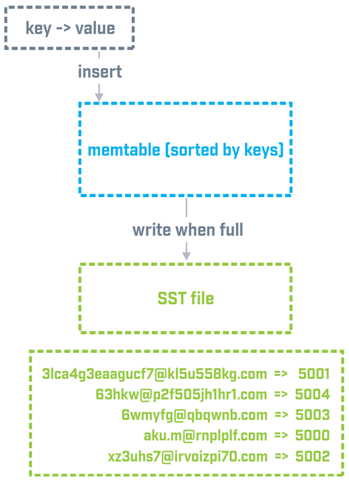
What does this look like? Using our previous examples, let's assume we have the following users registered:

```
(user_id: 5000; user_email: "aku.m@rnplplf.com")
(user_id: 5001; user_email: "3lca4g3eaagucf7@kl5u558kg.
com")
(user_id: 5002; user_email: "xz3uhs7@irvoizpi70.com")
(user_id: 5003; user_email: "6wmyfg@qbqwnb.com")
(user_id: 5004; user_email: "63hkw@p2f505jh1hr1.com)
```

After being sorted and written to disk, the email index looks something like:

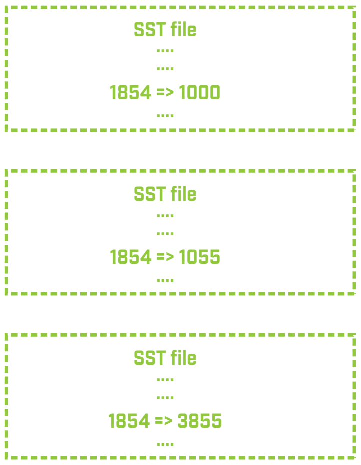
```
(key=>value)
3lca4g3eaagucf7@kl5u558kg.com => 5001
63hkw@p2f505jh1hr1.com => 5004
6wmyfg@qbqwnb.com => 5003
aku.m@rnplplf.com => 5000
xz3uhs7@irvoizpi70.com => 5002
```

This results in the entire buffer being written to memory in one sequential operation:


key -> value
insert
memtable (sorted by keys)
write when full
SST file
3lca4g3eaagucf7@kl5u558kg.com => 5001
63hkw@p2f505jh1hr1.com => 5004
6wmyfg@qbqwnb.com => 5003
aku.m@rnplplf.com => 5000
xz3uhs7@irvoizpi70.com => 5002

That's the benefit, but what are the drawbacks?

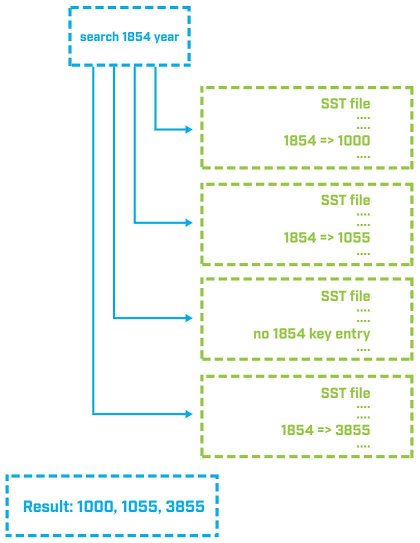
As we continue to insert users and write to the disk, LSM creates an increasing number of "SST files." Each of these files are sorted, and there is no global order. Moreover, the same key (for non-unique indexes) can end up in different files. The following diagram illustrates how SST files for "Year_of_birth" indexes might look:



This organization makes searching an individual file fast, but searching globally slow. For example, if we want to find the "user_id" for a user with email w7hl@125msxuyf7.com, we would need to look in each file individually.

This presents two problems:

- Searching data by an individual key
- Searching data by a range of keys (e.g., all users with "`year_of_birth`" between 1970 and 1990)



In order to address SST files' distributed nature, production software often implements different maintenance logic:

- **File compaction:** merging files into one
- **File levels:** making file hierarchies, to avoid checking each file for an existing key

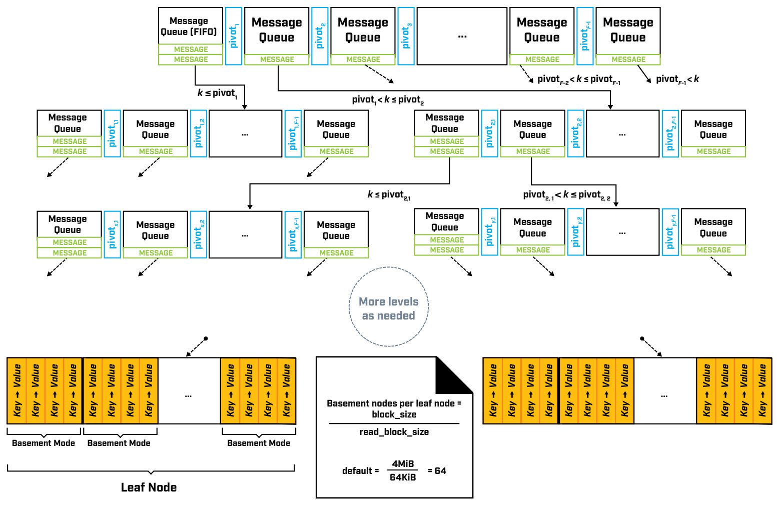
- **Bloom filters:** helps lookup individual keys faster (but doesn't help with ranges)

FRACTAL TREE

Fractal Tree data structures are closer to traditional B-tree structures—but instead of applying changes immediately, changes are buffered. As information exceeds the limits of the main index memory, the tree data structure buffers large groups of messages. The buffered data is slowly pushed down the tree as the buffers fill up. When data gets to a leaf node, there is a single IO applied to the data. This helps avoid random operations causing performance degradation by performing buffer changes all at once.

Data compression reduces read IO further.

The following diagram demonstrates the buffering process:



By combining all writes, Fractal Trees save time by performing a single transaction rather than a number of random ones. However, because a huge number of messages reside in the buffer, SELECT functions now must traverse through all the messages in order to find the correct one (and this is especially bad for point SELECT queries).

Remember: Primary key or unique key constraints require a HIDDEN POINT SELECT lookup! This means that both a UNIQUE KEY and a non-sequence PRIMARY KEY are performance killers for Fractal Tree data structures.

Fractal Trees are a good structure for databases with a lot of tables, indexes (preferably non-unique indexes), and a heavy write workload. It is also good for systems with slow storage times, or for saving space when the storage is fast but expensive.

Lastly, this is often a good fit for cloud-based database environments, where again storage is often slow (or if fast, expensive).

IMPLICATION FOR SLOW READS

Unfortunately for performance, LSM and Fractal Trees are less friendly for read operations.

Direct and implicit read operations are slower for LSM and Fractal Tree structures. Things like unique key constraints make insert and update transactions slower (because the background data is checked to see if the value exists).

Foreign key constraints will also slow down insert and update transactions on corresponding tables. Some schemas don't support foreign keys (or unique keys, for that matter).

Finally, select index and join operation transactions can also be affected. One way around this issue is to use covering indexes. A covering index contains all of or more of the columns you need for your query.

For example, let's assume you want to execute query:

```
SELECT user_name FROM users WHERE user_email='sherlock@
holmes.guru'
```

or in MongoDB notation:

```
db.users.find(  {user_email: "sherlock@holmes.guru" } ,
{user_name : 1} )
```

The index { user_email => user_id } results in two operations:

- Lookup user_id by user_email
- Lookup user_name by user_id

One solution instead is to create an index (in SQL syntax):

```
CREATE INDEX idx_email (user_email, user_name)
```

This query:

```
SELECT user_name FROM users WHERE user_email='sherlock@
holmes.guru'
```

can now be resolved by accessing the index idx_email, as "user_name" is now the part of index.

This "trick" can also be used with B-trees, but it works best with LSM and Fractal Trees for additional index overhead.

CONCLUSION

As we've discussed, the three data structure that can be used (B-tree, LSM tree, and Fractal Tree) can affect data performance in relation to applications. Using a database system based on one of these algorithms can affect the way your queries perform. The storage method affects the data performance—and data performance is a key component to application performance. Your business depends on application performance, and how your customers view how well your applications respond.

VADIM TKACHENKO is the CTO of Percona. He is an expert in LAMP performance, especially optimizing MySQL and InnoDB internals to take full advantage of modern hardware using his multi-threaded programming background. Vadim co-founded Percona in 2006 after four years in the High Performance Group within the official MySQL Support Team. He serves on Percona's Executive Team. He also co-authored the book, High Performance MySQL 3rd Edition,

A Survey of ORM Libraries for Android and iOS

QUICK VIEW

01
The choice of SQLite as a persistence layer for iOS is certainly a non-standard decision.

02
There is still no official JDBC driver for Android and SQLite.

03
Surprisingly, there aren't many ORM libraries which support both Android and iOS.

BY AGNIESZKA KOZUBEK-KRYCUŃ + PAWEŁ POSKROBKO

How do you persist data in a mobile application? Mobile platforms offer a variety of storage options: shared preferences, files, relational databases, network servers, and others. The choice of an appropriate storage option is not straightforward. Even if you decide to use a relational database, you still have to decide which API to use. In this article, we'll do a survey of available relational database libraries for Android and iOS.

SQLITE ON ANDROID AND IOS

SQLite is available and natively supported by all Android devices. It is also deployed on every iOS device. However, the choice of SQLite as a persistence layer for iOS is certainly a non-standard decision; the suggested choice is CoreData. In a typical configuration, CoreData is backed up by an SQLite database, but there is no direct access to an SQLite instance. When you develop the same application for Android and iOS in parallel, though, using the same persistence layer allows you to have similar software architecture on both platforms.

THE BENEFITS OF USING AN ORM

An Object-Relational Mapping (ORM) is a software library which—generally speaking—knows how to translate table rows into objects and vice versa. The discussion of whether you can write a really good ORM is as old as ORMs themselves. In spite of the controversies, ORMs are widely used in virtually all object-oriented languages.

What are the benefits of ORMs? First, they offer an object-oriented model of the database. The developers of an object-oriented language do not have shift paradigms each time they access the database, making their job easier. And while a query builder is not necessarily part of an ORM library, most mature ORMs come with an SQL query builder. ORMs also often help manage the database creation process and database schema changes.

ORM PATTERNS

Two of the most common patterns in ORM implementations are Active Record and Data Access Object. In the Active Record pattern, each table is represented as a class, while table rows are translated to the object of the corresponding class. The objects know how to persist themselves in a database.

In the Data Access Object pattern, database access is delegated to dedicated Data Access Objects (or DAOs). DAOs know how to persist each object and how to construct objects from the database.

ORM LIBRARIES IN ANDROID

Only four years ago, there were hardly any ORM libraries for Android. Nowadays, the number of ORM libraries for Android is constantly growing.

Since Java is the language you use to program an Android application, it seems natural to consider porting an existing Java ORM library to Android. The main problem with such a port is that there is still no official JDBC driver for Android and SQLite. (Some third-party JDBC drivers are available.) Moreover, mobile devices' limited memory and processor capabilities makes porting the all-powerful Hibernate infeasible.

GreenDAO is an open-source ORM library for Android developed by the German company greenrobot. First released in 2012 and still actively maintained, it is one of the most popular ORMs for Android. As its name suggests, GreenDAO uses the Data Access Object pattern. The entity classes are generated with a code generator in a separate Java project. With this approach, GreenDAO avoids the runtime processing of annotations (extremely time-consuming on mobile devices). This library has the reputation of being the fastest Android ORM. It has a fluent interface query builder; it's also thread-safe and supports transactions. In GreenDAO, database tables are created automatically but migrations between database schema versions have to be handled manually. An interesting feature of this ORM is its asynchronous API (still in the beta phase).

ORMLite is an open-source Java ORM which supports relational database engines like MySQL, PostgreSQL, SQL Server, SQLite (via JDBC), and others. It has been ported to Android with the help of low-level SQLite API calls. It was first released in 2011, making it one of the first popular Android ORMs. The latest activity in its Github repository was mid-2015.

Like GreenDAO, ORMLite follows the Data Access Object pattern, though the entity classes have to be coded manually. Their database representation comes with annotations; these are processed at runtime, which makes the library quite slow. The database tables have to be created manually, but there is a utility class to help things along. Schema migrations must be handled manually with ALTER statements. ORMLite has a fluent interface query builder.

DBFlow, first released in 2014, is one of the newest ORM libraries for Android, and it's gaining in popularity. DBFlow uses Active Record patterns, and database tables are created automatically. The entity classes are coded manually and are set up with annotations. DBFlow supports migrations and usage of multiple databases; it also offers functions like lazy loading, caching, observable models, and more. Its fluent interface query builder is powerful and supports JOINs and other advanced features.

ActiveAndroid (as the name suggests) uses the Active Record pattern. In fact, ActiveAndroid was the first Android ORM to use that pattern; however, it is no longer maintained. Its author, Michael Pardo, recommends switching to his new Android ORM, Ollie. (Ollie is not very popular yet.)

ActiveAndroid requires you to hand-code entity classes. They inherit from a Model class, and table and column properties are set using annotations. Database tables are created automatically. ActiveAndroid supports very simple migrations: new tables are added automatically, but changes in the existing tables have to be handled with a manually-created SQL script. ActiveAndroid has no real query builder.

SugarORM is another Android ORM that uses the Active Record pattern. Released in 2012, it is still regularly maintained. The entity classes have to be written manually; they inherit from the SugarRecord class or are set up with annotations. The database tables are created automatically by the ORM. SugarORM knows how to handle migrations, but actual SQL scripts have to be provided. This ORM has a fluent interface query builder.

DATABASE ACCESS LIBRARIES IN IOS

Since the standard persistence layer for the iOS platform is CoreData, there aren't many ORM (or even database) libraries for iOS. SQLite is written in C, so you can use its native API directly in an Objective-C application in iOS.

FMDB is the most popular still-actively-maintained SQLite library for iOS. Unlike all other libraries mentioned in this article, this is **not** an ORM library. FMDB is a wrapper over an SQLite API, and it provides some convenient functions. You can only execute raw queries—there is no query builder—but you can use binding syntax to prevent from SQL injection. There are many other libraries built on top of FMDB. One example (no longer maintained) is EGODatabase, a thread-safe version of FMDB.

DBAccess is a free, albeit closed-source, ORM library for iOS. It was first released in 2014, and it is still actively maintained. DBAccess uses the Active Record pattern, it has a fluent interface query builder, and it is thread-safe. Entity classes are coded manually by the developer, and database properties are set up using the appropriate @dynamic and @synthesize properties. DBAccess automatically creates the database tables.

DBAccess also offers more advanced features like JOINs or asynchronous queries. Its most interesting feature is events. These can register asynchronous blocks of code, which are executed after database events (i.e. inserts, updates, or deletes).

DATABASE LIBRARIES FOR BOTH ANDROID + IOS

Surprisingly, there aren't many ORM libraries which support both Android and iOS. One example of such a library is the newly-released **Vertabelo Mobile ORM**. This library uses the Data Access Object pattern. It has a unique approach to generating entity classes. You start with the database model in Vertabelo, a visual database modeling tool. The entity classes, DAO classes, and other runtime classes are generated from the diagram and downloaded as a zip file. The modeling tool lets you generate the SQL script file, which has to be run against the database. Vertabelo Mobile ORM has a fluent interface query builder.

The newest alternative to ORM libraries for Android and iOS, **Realm**, is rapidly gaining popularity. Realm is a mobile database, meant as a replacement for SQLite and CoreData. It is based on Realm Core, its own non-relational storage engine. Realm provides database access libraries for Android, Objective-C, and Swift. The libraries are free and open-source; the storage engine is currently closed-source, but the authors plan to open-source it.

Even though non-relational storage engines are an important part of the storage engines landscape for mobile platforms, there is a lot going on in the world of relational database access libraries. It is worth keeping an eye on both sides of this coin.

AGNIESZKA KOZUBEK-KRYCUŃ is the editor-in-chief of the Vertabelo blog. She has 7 years experience as a Java developer. Her most notable projects were OneWebSQL, a Java ORM offered by e-point SA, and Vertabelo, a database modeling tool. She holds a PhD in Mathematics and teaches programming courses at the University of Warsaw.

PAWEŁ POSKROBKO is a developer at Vertabelo and a student of Computer Science at the University of Warsaw. He was responsible for the development of the iOS version of Vertabelo Mobile ORM - experience and knowledge gained during the process of its development gave him an extensive view on the available database solutions for mobile platforms.

The No+SQL Database for Mission-Critical Data.

It’s not Magic, it’s FairCom.

FairCom®

www.FairCom.com

Bringing Data Into Focus

Remember that 3D art made famous with the trademark Magic Eye? That colorful abstract art appears to be nothing more than a rush of shapes and colors. Except there is more than meets the eye. If you focus in just the right way, suddenly—like magic—the art comes into focus and the meaning becomes clear, transforming your experience.

The emergence of NoSQL and IoT has brought on a rush of Big Data. It's unstructured. It's different data models and it's coming from everywhere. Big Data can feel a lot like Magic Eye. You don't know what you're looking at and you're going crossed-eyed trying to decipher the meaning. But the right tools can transform your experience and bring your data into focus.

By bringing data into focus companies can decipher meaning through BI for opportunities to monetize the data by delivering value to customers.

Transforming NoSQL with ACID – Very few NoSQL-oriented databases allow ACID-compliant transactions with unstructured data. Databases are starting to do more to allow flexibility in their handling of ACID properties. NoSQL has already shown us the advantages of a relaxed C mode—most NoSQL systems are eventually consistent. Yet, many are starting to offer tunable consistency. At FairCom we innovated with durability to allow for a relaxed D mode.

Transforming NoSQL with SQL – Combining two or more data models in a single database management system will allow an organization to expand the data types an application can handle. FairCom is unique in the way we handle multi-schema data—our NoSQL and SQL APIs run on the same datasets. FairCom allows analytical applications to access unstructured, NoSQL data through relational interfaces.

It's not magic that brings your data into focus—it's the right tools.

WRITTEN BY EVALDO HORN DE OLIVEIRA

DIRECTOR OF BUSINESS DEVELOPMENT AT FAIRCOM

c-treeACE BY FAIRCOM CORPORATION

FairCom

The kicker is that the capabilities are multimodel rather than just relational. From the developer's perspective, what you see is flexibility.

CATEGORY	NEW RELEASES	OPEN SOURCE?
Multimodel NoSQL + SQL	Version 11 November 2015	No

CASE STUDY

FairCom's innovation is in its adoption of a multimodel approach to databases, which currently embraces both the relational model and a variety of NoSQL capabilities, employing virtual tables to deliver SQL access to unstructured data. The c-treeACE multimodel capabilities are built in at the physical level. The database is ACID compliant, irrespective of the mode of data access and update or the specific API that is used. From a relational perspective, c-treeACE looks very much like a fully-functional relational database. The kicker is that the capabilities are multimodel rather than just relational. From the developer's perspective, what you see is flexibility.

STRENGTHS

- An advanced, multimodel database offering an ACID-compliant, key-value store and full SQL engine operating on the same data
- Nearly zero-administration database, ideal for ISVs
- High availability through hot backups, replication, and auto-recovery
- Unprecedented performance through sophisticated tuning facilities
- Advanced security and encryption capabilities

NOTABLE CUSTOMERS

- Visa
- Totvs
- UPS
- Tealeaf
- ACI Worldwide
- CA

BLOG www.faircom.com/developers

TWITTER @faircom_corp

WEBSITE faircom.com

How to Choose a DBaaS

BY WILL SHULMAN

QUICK VIEW

01
Managing application infrastructure, particularly the database tier, is hard.

02
DBaaS (Database-as-a-Service) is a new category of cloud infrastructure that abstracts away all of the complexities of database management.

03
Selecting the right DBaaS provider for your application can dramatically reduce the time and effort required to build and run your production applications.

With the rise of cloud infrastructure, cloud-based services are available for almost every component of the modern application stack. The database layer is no exception.

Called DBaaS (Database-as-a-Service), cloud database services exist for almost all of the modern relational databases (MySQL, Postgres, etc.), as well as for NoSQL databases such as MongoDB, CouchDB, and Neo4J.

But what is a DBaaS and how should you go about evaluating all of the various DBaaS platforms out there?

This article will explain DBaaS and outline all of the important characteristics one must consider when hosting a production database in the cloud.

WHAT IS A DBAAS?

DBaaS providers host your database infrastructure and handle all of the low-level operational aspects of managing your database so that you can focus on application development.

To achieve this, DBaaS providers not only host your database software and your data, but also manage all of the hardware and networking infrastructure beneath it. They also automate all management activities such as provisioning, scaling, failover, and backup / restore, as well as offer support for when you need help.

HOW TO SELECT A DBAAS

First, you want to make sure you have selected your database technology. Evaluating DBaaS providers ahead of this step would be putting the cart before the horse. You should select the right database technology to meet your application's technical requirements and then seek out a DBaaS provider for that database technology.

Once you know what database(s) you will be using, you will want to consider the following when assessing DBaaS providers.

DATABASE LOCATION

Not all cloud services need to be physically proximate to your application servers, but your database layer does. This is for two reasons:

- **Latency:** You want to minimize the amount of time it takes to send a request to your database and get a response, as this latency has a huge impact on overall application performance.
- **Security:** Ideally the network between your application and your database is private and data is not travelling over the open internet.

This is why, for the majority of applications, you should place your application servers and your database servers (via your DBaaS provider) in the same datacenter.

For example, if your application tier is in Microsoft Azure's West US datacenter you want a DBaaS that can provision your database in Azure West US.

Also, you should consider the extent to which your DBaaS provider locks you into a particular cloud or geographic region. A provider that offers a variety of clouds to run on (e.g., AWS, Azure, Google), can give you peace of mind that you will be able to change cloud providers, or use multiple cloud providers, without needing to change DBaaS solutions.

FAULT TOLERANCE, AVAILABILITY + REDUNDANCY

If you are running a production application, your database should always be available, even in the face of hardware failure and maintenance. Your DBaaS provider is instrumental in making High Availability a practical reality, regardless of your underlying database technology.

To achieve fault tolerance, DBaaS providers usually provide multi-node database clusters that can withstand node failures.

Things to consider:

- Does the provider offer fault tolerance via clustering?
- If so, how isolated are the nodes in the database cluster? Some providers simply spread database nodes across multiple racks in the same datacenter while others have more physically isolated zones such (i.e., AWS Availability Zones). Proper isolation is critical to minimizing the likelihood of downtime.
- How does system failover work? Is it automatic or do you have to intervene?

- How are faulty components replaced? Is it automatic or do you have to intervene?
- Does the provider offer an availability SLA?
- Does the provider offer global disaster recovery (DR) in the event of a regional datacenter outage?

DATA DURABILITY + BACKUPS

Your DBaaS provider should have a robust backup and recovery system. You must ensure that you can recover from a catastrophic failure and, more likely, human error (e.g., a developer accidentally deletes data).

Things to consider:

- Does the provider automatically take backups of your data?
- Does the provider have tools for managing backups?
- Can you create recurring backup plans to automatically take backups on a custom schedule?
- Can you easily and quickly restore from backup?
- Does the provider support point-in-time restores, allowing you to restore to any time in the past or only to the time of last backup?

MONITORING, PERFORMANCE ANALYTICS + ALERTING

You need to ensure that your database is always available and fast. Monitoring, performance analytics, and alerting features that give you insight into the health of your database deployment are crucial.

Your DBaaS provider should provide both uptime and performance monitoring, with the ability to generate alerts that get delivered to your team if any important metrics are outside normal operating range.

Things to consider:

- Does the provider automatically alert you when there is a component failure in your deployment?
- Does the provider offer real-time insight into database performance metrics?
- Does the provider offer historical reporting of database performance metrics?
- Does the provider let you create custom alerts based on database performance metrics?
- Does the provider support easy access to database log files?

PERFORMANCE + SCALING

Your provider should provide a platform that not only performs well for your workload but also can scale to maintain that performance as your data volume grows.

If your application has demanding performance requirements, the best way to assess the service is to test it with your workload. This means, to the best of your abilities, you should try to simulate the operation mix and load that will come from your production app. Beware of benchmarks, as they are usually so particular to the workload being tested that they may not give a good picture of how the service will perform for you.

If you expect significant growth in either the volume of your data or the amount of database traffic your application generates, you will want to make sure the provider makes it easy to scale.

There are two general techniques to scaling. One is vertical scaling, where you add resources (RAM, CPU, Disk) as your deployment grows. The second is horizontal scaling, where you add more nodes to the system to handle the growth in data volume and / or database traffic.

Ultimately, if you plan to have a relatively large dataset (hundreds of GBs), you will want to make sure your provider has a solution that scales horizontally. While vertical scaling can be very effective, and even preferred at small scale, there is a limit to how much hardware can fit in one box. Horizontal scaling is essential for larger deployments.

SECURITY

Your data may be the single most important asset of your business. As such, your DBaaS provider must be expert in security and able to provide you with tools to ensure that your data is secure from unauthorized access.

Things to consider:

- Is authentication required to connect to your cloud-hosted database?
- Is all access to the database logged?
- Can you configure firewalls so that only your application has network access to your database?
- Does the provider support communicating with your database via SSL with certificate validation?
- Does the provider support encryption at rest?
- Does the management interface you use to manage your cloud database deployment support two-factor authentication?
- Does the provider undergo third-party penetration testing and security audits to ensure they follow security best practices?
- Does the provider have any security and compliance certifications that are required for your organization, such as HIPAA?

SUPPORT

Fast, helpful support is a crucial component to ensuring your database and your application run smoothly. Your provider must offer great support, particularly when giving advice and responding to emergencies.

Things to consider:

- Does the provider offer support as part of the subscription or is it an additional fee?
- Does the provider offer premium or emergency support with guaranteed quick response times?
- Does the provider offer an SLA around support response times?
- Is the support actually thoughtful and helpful? (Contact them with database or vendor-specific questions and see how timely and helpful their response is.)
- Does the provider have a good reputation for outstanding support?

CONCLUSION

The right DBaaS provider can be an invaluable partner, but finding the right cloud service for your application requires research and forethought. Hopefully this mini-guide will help you frame your investigation and offer some guidance on what to look for.

WILL SHULMAN is CEO/Co-founder of mLab, a DBaaS for MongoDB, and is a technologist and entrepreneur with over 15 years of experience in building innovative software platforms and products. Before mLab, he was CTO and co-founder of Merced Systems, a high-growth enterprise analytics software company serving Global 2000 customers in more than 20 countries worldwide (acquired by NICE Systems for $190 million). Will holds a B.S. in Computer Science from Stanford University.

FINDING THE DATABASE FOR YOUR USE CASE

DB TYPE	STRONG USE CASES	WEAK USE CASES
Relational DB **Examples:** MySQL, PostgreSQL, SQL Server	• When ACID transactions are required (also a feature of several NoSQL DBs, e.g. Neo4j) • Looking up data by different keys with secondary indexes (also a feature of several NoSQL DBs) • When strong consistency for results and queries is required • Conventional online transaction processing • Risk-averse projects seeking very mature technologies and widely available skills • Products for enterprise customers more familiar with relational DBs	• Systems that need to tolerate partition failures • Schema-free management • Handling any complex / rich entities that require you to do multiple joins to get the entire entity back • Large changes in scale are predicte
Key-Value Store **Examples:** Redis, Riak, DynamoDB	• Handling lots of small, continuous, and potentially volatile reads and writes (also look for any DB with fast in-memory access or SSD storage) • Storing session information, user preferences, configurations, and e-commerce carts • Simplifying the upgrade path of your software with the support of optional fields, adding fields, and removing fields without having to build a schema migration framework	• Correlating data between different sets of keys • Saving multiple transactions (Redis is exempt from this weakness) • Performing well during key searches based on values (DynamoDB is exempt) • Returning only partial values is required
Document Store **Examples:** MongoDB, Couchbase, RavenDB	• Handling a wide variety of access patterns and data types • Handling reads with low latency • Handling frequently changing, user generated data • Simplifying the upgrade path of your software with the support of optional fields, adding fields, and removing fields without having to build a schema migration framework • Rapid prototyping • Blogs, profiles, and other entities that don't require relationships • Deployment on a mobile device (Mobile Couchbase)	• Atomic cross-document operations (RavenDB is exempt) • Querying large aggregate data structures that frequently change • Returning only partial values is required • Partial updates of documents (especially child/sub-documents) • Joins are desired • Foreign key usage is desired
Column Store **Examples:** Cassandra, HBase, Accumulo	• When high availability is crucial, and eventual consistency is tolerable • Event Sourcing • Logging continuous streams of data that have no consistency guarantees • Storing a constantly growing set of data that is accessed rarely • Deep visitor analytics • Handling frequently expiring data (Redis can also set values to expire)	• Early prototyping or situations where there will be significant query changes (high cost for query changes compared to schema changes) • Referential integrity required • Processing many columns simultaneously
Graph Store **Examples:** Neo4j, Titan, Giraph	• Handling entities that have a large number of relationships, such as social graphs, networks, tag systems, or any link-rich domain • Routing and location services • Recommendation engines or user data mapping • Dynamically building relationships between objects with dynamic properties • Allowing a very deep join depth • MDM solutions, CMDBs	• High volume write situations • Serving and storing binary data • Querying unrestricted across massive data sets • Storing large and/or orphaned, disconnected documents

Sources: *NoSQL Distilled, High Availability*
Edited by *Duncan Brown*

DIVING DEEPER

INTO DATA PERSISTENCE

TOP 10 #DATA TWITTER FEEDS

@MERV

@BRENTO

@SQLNIKON

@PINALDAVE

@AL3XANDRU

@KELLABYTE

@MYSQL

@EMILEIFREM

@SQLPERFTIPS

@SVE_SIC

DZONE DATA-RELATED ZONES

Database Zone

dzone.com/database

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

Big Data Zone

dzone.com/big-data

The Big Data/Analytics 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.

Performance Zone

dzone.com/performance

Scalability and optimization are constant concerns for the developer and operations manager. The Performance Zone focuses on all things performance, covering everything from database optimization to garbage collection, tool and technique comparisons, and tweaks to keep your code as efficient as possible.

TOP DATABASE REFCARDZ

Database Partitioning With MySQL bit.ly/mysqldatapart

NoSQL and Data Scalability
bit.ly/nosql20scalability

Essential PostgreSQL
bit.ly/essentialpostgresql

TOP DATABASE WEBSITES

DB-Engines db-engines.com

Database Trends and Applications dbta.com

Database Journal
databasejournal.com

TOP DATABASE RESOURCES

The Real World of the Database Administrator
bit.ly/DBAdminTrends

Fundamentals of Database Management Systems
bit.ly/DBMgmtSys

Executive Insights on Data Persistence

BY TOM SMITH

QUICK VIEW

01
The number and diversity of specialization of databases has increased exponentially over the recent past.

02
Applications are tending to use multiple databases to provide polyglot persistence of data.

03
The future of databases is the continuing growth of data and the demand for real-time analysis and predictive analytics on the edge.

In order to gauge the state of the Persistent Data and Databases in the "real world," we interviewed 16 executives, from 13 companies, actively involved in databases and persistent data. All of the executives have extensive experience in data and data management.

Satyen Sangani	CEO, ALATION
Sam Rehman	CTO, ARXAN
Andy Warfield	CO-FOUNDER/CTO, COHO DATA
Rami Chahine	V.P. PRODUCT MANAGEMENT, DATAWATCH
Dan Potter	CMO, DATAWATCH
Eric Frenkiel	CO-FOUNDER/CEO, MEMSQL
Will Shulman	CEO, MLAB
Philip Rathle	V.P. OF PRODUCT, NEO TECHNOLOGY
Paul Nashawaty	PRODUCT MARKETING AND STRATEGY, PROGRESS
Joan Wrabetz	CTO, QUALI
Yiftach Shoolman	CO-FOUNDER AND CTO, REDIS LABS
Leena Joshi	V.P. PRODUCT MARKETING, REDIS LABS
Partha Seetala	CTO, ROBIN SYSTEMS
Dale Lutz	CO-FOUNDER, SAFE SOFTWARE
Paul Nalos	DATABASE TEAM LEAD, SAFE SOFTWARE
Jon Bock	V.P. OF PRODUCT AND MARKETING, SNOWFLAKE COMPUTING

Here's what we learned from the executives:

01

Companies tend to use their own databases as well as those their clients are using. Service providers are agnostic with regards to the databases they use. They also have a good understanding of the specific strengths of each database. Specific mentions of non-proprietary databases included: MongoDB, Cassandra, Spark SQL, MySQL, PostgreSQL, Teradata, Vertica, Oracle, AWS RDS for Aurora, Geodatabase, Smallworld, and even Microsoft Excel.

02

There's a consistent definition of persistent data as **data that doesn't change across time, systems, and memory**; data that's considered durable at rest with the coming and going of software and devices; master data that's stable, that is set and recoverable whether in flash or in memory.

03

The most important elements of the database depend on what is needed. **Foremost is storing data in some form of durability to maintain asset properties with the ability to access it**. There's a tradeoff between speed, scale, and usability. Ultimately databases must be consistent, available, and able to tolerate partitions. The ability to support a broad variety of data for aggregation, analysis, and reporting. Performance, scalability, and the ability to process more data more quickly is becoming more important as data becomes more prolific. Databases have bifurcated into what's most relevant for the use case—"the consumerization of databases."

There are six features critical to ensuring high availability and safeguard against every type of failure or outage event: 1) in-memory replication; 2) multi-rack/zone/data center replication; 3) instant auto-failover; 4) AOF (append-only file) data persistence; 5) backup; and 6) multi-region/cloud replication.

04

Databases are enabling companies to use data to inform real-time decisions about their business as well as to use predictive analytics to **make better informed, real-time decisions**. The macro-trend is that more data is being analyzed in real-time. The internet of connected things enables you to see how things interact. Applications are tending to use multiple databases to provide polyglot persistence.

05

There were a **number of skills** mentioned by executives that make someone good at working with databases. These include: understanding the proper design structure, knowing what's in the database you're working with, and understanding data science and what data scientists are looking for. As the number of databases grow, it's important to understand the strengths and weaknesses of the different tools and to choose the right database for what you're trying to accomplish. More Big Data jobs are requiring a broader set of skills.

06

Data management has **evolved very rapidly since the introduction of Hadoop and Big Data**. We've gone from gigabytes to zettabytes of data that is distributed across—and needs to be accessed from—several sources very quickly. Organizations are building cultures with data scientists and data management now has visibility in the C-suite with the Chief Data Officer reporting to the CEO or the CTO.

07

The obstacles to success are consistent with the growth of data and the growth of databases. Data resides in a number of different places and you need access to the data sources regardless of where they are. There's an **explosion of new database technologies**, and someone in the organization needs to stay abreast of what's available and what's the best solution to the problem at hand, someone with more diverse data literacy with different databases and languages. Given the growth and variety of options, it's rare for an enterprise to have the resources they need to analyze Big Data themselves. This has led to the growth of companies providing databases as a service (DBaaS), since these companies have the bandwidth to keep up with all of the latest technologies, know their strengths and weaknesses, and employ professionals who know the nuances of each database.

08

The only concerns around data management are the **tremendous growth in the number of databases and the inherent complexity therein**. Several people expressed concern that it's more complex than it needs to be, as marketers create confusion with different terminology and have a tendency to overpromise and under-deliver. There's agreement that many of the database options will coalesce over time, and there will be SQL and NoSQL options—with those in the know realizing that all NoSQL databases are not the same.

09

The future for databases involves **consolidation around Big Data down to around 10 core technologies** that make data easy to access and leads to more data-driven analytics and services. Data will be easier to access and use. More processing will be done on the edge to facilitate real-time computations and decision making. Polyglot persistence will ensure the safety of persistent data. Data science will improve research by defining the questions that need to be asked.

10

What developers need to keep in mind when working with different databases is consistent with what they need to keep in mind when working with all technologies: use **best practices that are already established, proven, and tested**; don't reinvent the wheel if you already have the right technology for the job; understand how the data will be used so it's in the right data store and language for the required analysis; and, there's no such thing as "one size fits all," so don't become too attached to a single solution. Specific recommendations include: knowing SQL while learning as many other languages as you can; exploring JSON; getting up to speed on predictive analytics; and considering geospatial data, given the growth of mobile.

11

Other trends mentioned by the executives are the **importance of supporting SQL**, and understanding when one database is more cost efficient than another as the data quickly scales. Lastly, open source, and the role it plays, is a very big trend since open source has democratized the database layer.

The executives we spoke with are fully invested in the evolution of databases and data management and want to continue to lead its evolution and success to meet business and consumer needs. We're interested in hearing from developers, and other IT professionals, to see if these insights offer real value. Is it helpful to hear others' perspectives from an executive point of view? Are their experiences and perspectives consistent with yours?

We welcome your feedback at research@dzone.com.

TOM SMITH is a Research Analyst at DZone who excels at gathering insights from analytics—both quantitative and qualitative—to drive business results. His passion is sharing information of value to help people succeed. In his spare time, you can find him either eating at Chipotle or working out at the gym.

SOLUTIONS DIRECTORY

This directory contains databases and database performance tools to help you store, organize, and query the data you need. It provides free trial data and product category information gathered from vendor websites and project pages. Solutions are selected for inclusion based on several impartial criteria, including solution maturity, technical innovativeness, relevance, and data availability.

PRODUCT	CATEGORY	OPEN SOURCE?	WEBSITE
ActiveObjects by Atlassian	ORM	Included in Jira and Confluence	atlassian.com
ActiveRecord	ORM	Included in Rails	rubyonrails.org
Adabas by Software AG	Stream Processing	Free tier available	softwareag.com
Aerospike Server	In-Memory, KV	Open source	aerospike.com
Altibase HDB	In-Memory, NewSQL	Free tier available	altibase.com
Apache Cassandra	KV, Wide Column	Open source	cassandra.apache.org
Apache Hbase	Wide Column	Open source	hbase.apache.org
Apache Ignite	In-Memory, Hadoop, Data Grid	Open source	ignite.apache.org
Apache OpenJPA	ORM	Open source	openjpa.apache.org
ArangoDB	Graph, Document, KV	Open source	arangodb.com
Aster Database by Teradata	Specialist Analytic	Available by request	teradata.com
c-treeACE by FairCom	NewSQL, KV Direct Access	Available by request	faircom.com
Cache by Intersystems	Object-Oriented	Free tier available	intersystems.com
CakePHP	ORM	Open source	cakephp.org
ClustrixDB	NewSQL	Available by request	clustrix.com

PRODUCT	CATEGORY	OPEN SOURCE?	WEBSITE
Core Data by Apple	ORM	Included in iOS and OS X	developer.apple.com
Couchbase Server	KV, Document, Data Caching	Open source	couchbase.com
Django ORM	ORM	Open source	djangoproject.com
DynamoDB by Amazon	KV, DBaaS	Free tier available	aws.amazon.com
EclipseLink	ORM	Open source	eclipse.org
EDB Postgres Advanced Server by EnterpriseDB	RDBMS	Free tier available	enterprisedb.com
Entity Framework	ORM	Part of .NET Framework	msdn.microsoft.com
Hazelcast	In-Memory, Data Grid	Open source	hazelcast.com
Hibernate by Red Hat	ORM	Open source	hibernate.org
IBM DB2	RDBMS	Free tier available	ibm.com
In-Memory Data Fabric by GridGain	In-Memory, Hadoop, Data Grid	Free tier available	gridgain.com
Infinispan by Red Hat	In-memory, KV, Data Grid and Cache	Open Source	infinispan.org
Ingres by Actian	RDBMS	30-day free trial	actian.com
InterBase by Embarcadero	RDBMS	Free tier available	embarcadero.com
JDBC by Oracle	Java API	Part of Java SE	oracle.com
jOOQ	SQL Mapper	Open source	jooq.org
MariaDB	RDBMS, MySQL Family	Open source	mariadb.com
MemSQL	In-Memory, NewSQL	Free tier available	memsql.com
MongoDB	Document	Open source	mongodb.org
MyBatis	SQL Mapper	Open source	mybatis.org
MySQL Community Edition by Oracle	RDBMS	Open source	mysql.com
Neo4j by Neo Technology	Graph	Free tier available	neo4j.com

PRODUCT	CATEGORY	OPEN SOURCE?	WEBSITE
Nhibernate	ORM	Open source	nhibernate.info
NuoDB	NewSQL	Free tier available	nuodb.com
Oracle Database	RDBMS, Graph, Document	Free solution	oracle.com
OrientDB	RDBMS, Document, Graph	Open source	orientechnologies.com
OrmLite	ORM	Open source	ormlite.com
Percona Server	RDBMS, MySQL Family	Free solution	percona.com
Pivotal Gemfire	In-Memory, Data Grid	Free solution	pivotal.io
PostgreSQL	Object-Relational	Open source	postgresql.org
RavenDB by Hibernating Rhinos	Document	Open source	ravendb.net
Redis	In-Memory, KV, Data Caching	Open source	redislabs.com
Redis Labs Enterprise Cluster	In-Memory, KV, Data Caching	Free tier available	redislabs.com
Riak by Basho	Document, KV	Open source	basho.com
SAP HANA Platform	In-Memory, Column-Oriented RDBMS	Available by request	hana.sap.com
ScaleOut StateServer	In-Memory, Hadoop, Data Grid	30-day free trial	scaleoutsoftware.com
Splice Machine	NewSQL, Hadoop	Free tier available	splicemachine.com
SQL Server 2016 by Microsoft	RDBMS	180-day preview available	microsoft.com
SQLAlchemy	ORM	Open source	sqlalchemy.org
SQLite	RDBMS	Open source	sqlite.org
Storm by Canonical	ORM	Open source	storm.canonical.com
Toplink by Oracle	ORM	Free solution	oracle.com
Versant Object Database by Actian	Object-Oriented	30-day free trial	actian.com
VoltDB	In-Memory, NewSQL	30-day free trial	voltdb.com

glossary

AGGREGATE A cluster of domain objects that can be treated as a single unit. An ideal unit for data storage on large distributed systems.

ACID (ATOMICITY, CONSISTENCY, ISOLATION, DURABILITY) A term that refers to the model properties of database transactions, traditionally used for SQL databases.

BASE (BASIC AVAILABILITY, SOFT STATE, EVENTUAL CONSISTENCY) A term that refers to the model properties of database transactions, specifically for NoSQL databases needing to manage unstructured data.

B-TREE A data structure in which all terminal nodes are the same distance from the base, and all nonterminal nodes have between n and 2n subtrees or pointers. It is optimized for systems that read and write large blocks of data or perform mostly reads.

COMPLEX EVENT PROCESSING An organizational process for collecting data from multiple streams for the purpose of analysis and planning.

DATABASE CLUSTERING Connecting two or more servers and instances to a single database, often for the advantages of fault tolerance, load balancing, and parallel processing.

DATA MANAGEMENT The complete lifecycle of how an organization handles storing, processing, and analyzing datasets.

DATA MINING The process of discovering patterns in large sets of data and transforming that information into an understandable format.

DATABASE MANAGEMENT SYSTEM (DBMS) A suite of software and tools that manage data between the end user and the database.

DATA WAREHOUSE A collection of accumulated data from multiple streams within a business, aggregated for the purpose of business management.

DISTRIBUTED SYSTEM A collection of individual computers that work together and appear to function as a single system. This requires access to a central database, multiple copies of a database on each computer, or database partitions on each machine.

DOCUMENT STORE A type of database that aggregates data from documents rather than defined tables and is used to present document data in a searchable form.

EVENTUAL CONSISTENCY The idea that databases conforming to the BASE model will contain data that becomes consistent over time.

FAULT-TOLERANCE A system's ability to respond to hardware or software failure without disrupting other systems.

GRAPH STORE A type of database used for handling entities that have a large number of relationships, such as social graphs, tag systems, or any link-rich domain; it is also often used for routing and location services.

HADOOP An Apache Software Foundation framework developed specifically for high-scalability, data-intensive, distributed computing. It is used primarily for batch-processing large datasets very efficiently.

HIGH AVAILABILITY (HA) Refers to the continuous availability of resources in a computer system even after component failures occur. This can be achieved with redundant hardware, software solutions, and other specific strategies.

IN-MEMORY As a generalized industry term, it describes data management tools that load data into RAM or flash memory instead of hard-disk or solid-state drives.

JOURNALING Refers to the simultaneous, real-time logging of all data updates in a database. The resulting log functions as an audit trail that can be used to rebuild the database if the original data is corrupted or deleted.

KEY-VALUE STORE A type of database that stores data in simple key-value pairs. They are used for handling lots of small, continuous, and potentially volatile reads and writes.

LIGHTNING MEMORY-MAPPED DATABASE (LMDB) A copy-on-write B-Tree database that is fully transactional, ACID compliant, small in size, and uses MVCC.

LOG-STRUCTURED MERGE (LSM) TREE A data structure that writes and edits data using immutable segments or runs that are usually organized into levels. There are several strategies, but the first level commonly contains the most recent and active data.

MAPREDUCE A programming model created by Google for high scalability and distribution on multiple clusters for the purpose of data processing.

MULTI-VERSION CONCURRENCY CONTROL (MVCC) A method for handling situations where machines simultaneously read and write to a database.

NON-FIRST NORMAL FORM QUERY LANGUAGE (N1QL) Developed by Couchbase, it offers a common query language and JSON-based data model for distributed document-oriented databases.

NEWSQL A shorthand descriptor for relational database systems that provide horizontal scalability and performance on par with NoSQL systems.

NOSQL A class of database systems that incorporates other means of querying outside of traditional SQL and does not use standard relational structures.

OBJECT-RELATIONAL MAPPER (ORM) A tool that provides a database abstraction layer to convert data between incompatible type systems using object-oriented programming languages instead of the database's query language.

PERSISTENCE Refers to information from a program that outlives the process that created it, meaning it won't be erased during a shutdown or clearing of RAM. Databases provide persistence.

POLYGLOT PERSISTENCE Refers to an organization's use of several different data storage technologies for different types of data.

RELATIONAL DATABASE A database that structures interrelated datasets in tables, records, and columns.

REPLICATION A term for the sharing of data so as to ensure consistency between redundant resources.

SCHEMA A term for the unique data structure of an individual database.

SHARDING Also known as "horizontal partitioning," sharding is where a database is split into several pieces, usually to improve the speed and reliability of an application.

STRONG CONSISTENCY A database concept that refers to the inability to commit transactions that violate a database's rules for data validity.

STRUCTURED QUERY LANGUAGE (SQL) A programming language designed for managing and manipulating data; used primarily in relational databases.

WIDE-COLUMN STORE Also called "BigTable stores" because of their relation to Google's early BigTable database, these databases store data in records that can hold very large numbers of dynamic columns. The column names and the record keys are not fixed.