





DZone (/) > Database Zone (/database-sql-nosql-tutorials-tools-news) > 8 Rules For Creating Useful Database Indexes

8 Rules For Creating Useful Database Indexes


(/users/196910/anghelleonard.html) by **Anghel Leonard** (/users/196910/anghelleonard.html)  MVB </> CORE · Oct. 06,

20 · Database Zone (/database-sql-nosql-tutorials-tools-news) · Tutorial

 Like (3)
 Comment (0)
 Save
 Tweet

Multi-Document ACID Transactions Made Easy for NoSQL



With Couchbase, you can perform multi-document ACID transactions at the database layer using the same semantics as a relational database. Just one more reason (the easiest, least disruptive way to

Creating an index can be done via the SQL `CREATE INDEX` or programmatically via JPA 2.1 or Hibernate-specific annotations.

JPA 2.1 @Index

Starting with JPA 2.1, we can easily create indexes via the `@Index` annotation as in the following example:

Java

```

1 import javax.persistence.Column;
2 import javax.persistence.Entity;
3 import javax.persistence.Index;
4 import javax.persistence.Table;
5
6 @Entity
7 @Table( name = "author",
8     indexes = {
9         @Index(
10             name = "index_name",
11             columnList="name",
12             unique = true
13         ),
14         @Index(
15             name = "index_genre",
16             columnList="genre",
17             unique = false
18         )
19     })
20 public class Author{
21
22     @Column(name = "name", nullable = false)
23     private String name;
24
25     @Column(name = "genre", nullable = false)
26     private String genre;
27 }

```

Or, for defining multi-column indexes follow this example:

Java



```

1 import javax.persistence.Column;
2 import javax.persistence.Entity;
3 import javax.persistence.Index;
4 import javax.persistence.Table;
5

```

REFCARDZ (/refcardz) RESEARCH (/research) WEBINARS (/webinars) ZONES ▾

```

6 @Entity
7 @Table(
8     name = "author",
9     indexes = {
10         @Index(
11             name = "index_name_genre",
12             columnList="name, genre",
13             unique = true
14         )
15     })
16 public class Author {
17
18     @Column(name = "name", nullable = false)
19     private String name;
20
21     @Column(name = "genre", nullable = false)
22     private String genre;
23 }

```

Hibernate ORM provides a deprecated `org.hibernate.annotations.Index`, therefore rely on JPA 2.1 approach.

Ideally, we create indexes for optimizing the performance of our database and SQL queries. We create super-fast data access paths for avoiding scanning the tablespace. But, that's easy to say and hard to execute properly. What is the best set of indexes for your tables? How to decide that an index is needed? How to decide if an index is useless? Well, these are hard questions and the answers are tightly coupled to what queries you execute. Nevertheless, let's highlight a developer dedicated guideline that applies in most of the cases.

1. Don't Guess the Indexes

Over the years, I saw the following bad practice for creating database indexes: watch the tables (schema), and without knowing how these tables will be accessed, try to guess what are the proper indexes. It's like trying to guess the queries that will be executed, and most of the time, the results don't have decent accuracy.

As a rule of thumb, to create the proper set of indexes try to:

- get the list of SQL queries to be used
- estimate the frequency of each SQL query
- try to score the importance of each SQL query

Having these three coordinates, find the proper set of indexes that bring the highest optimizations and the smallest trade-offs.

2. Prioritize for Indexing the Most Used SQL Queries

Mainly, this step highlights the second bullet from above, *estimate the frequency of each query*. The most used SQL queries should have a major priority for indexing. If the most used SQL queries are optimized, then there are big chances to assure optimal application performance.

As a rule of thumb, create indexes for the most used (heavily exploit) SQL queries and build indexes based on predicates.

3. Important SQL Queries Deserve Indexes

When we talk about query importance, we primarily consider the importance of the query for the business and, secondary, the user importance. For example, if a query is run every day for banking transactions or is run by an important user (e.g., CIO/CDIO), it might deserve its own index. But, if a query is just a simple routine or is executed by a clerk then the existing indexes should provide the proper optimizations. Of course, this is not a rule for weighing

4. Avoid Sorting Operations by Indexing Group By and Order By

Calling SQL clauses such as `GROUP BY` and `ORDER BY` may invoke sorting operations. These kinds of operations are typically slow (resource-intensive operations) and therefore prone to add performance penalties (e.g., as `ORDER BY` does in SQL queries specific to pagination).

By indexing on the columns specified in `GROUP BY` and `ORDER BY` we can take advantage of optimizations that avoids sorting operations (since an index provides an ordered representation of the indexed data - keeps data preordered). Instead of applying sorting operations, the relational database may use the index. Here it is an example:

SQL

```
1 SELECT * FROM book
2 WHERE genre = "History"
3   AND (publication_date, id) < (prev_publication_date, prev_id)
4 ORDER BY publication_date DESC, id DESC
5 LIMIT 50;
```

To optimize this query, we can create an index as follows:

```
CREATE INDEX book_idx ON book (publication_date, id);
```

Or, even better:

```
CREATE INDEX book_idx ON book (genre, publication_date, id);
```

This time, the database uses the index order and doesn't use the explicit sort operation.

5. Rely on Indexes for Uniqueness

The most database requires unique indexes for primary keys and unique constraints. These requirements are part of schema validation. Striving to write your SQL queries around these required indexes brings important benefits.

6. Rely on Indexes for Foreign Keys

As the previous step mention, a primary key constraint requires a unique index. This index is automatically created, therefore the parent table's side takes advantage of indexing. On the other hand, a foreign key is a column (or combination of columns) that appears in the child table and is used to define a relationship and ensure the integrity of the parent and child tables.

It is highly recommended to create an index on each foreign key constraint on the child table.

While the unique index for the primary key is automatically created, the unique index for the foreign key is the responsibility of the database administrator or the developers. In other words, if the database doesn't automatically create indexes for the foreign keys (e.g., SQL Server) then the indexes should be created manually by the database administrator or the developers.

Among the benefits of using indexes for foreign keys we have:

- calling the indexed foreign key on your SQL `JOIN` between the child and the parent table columns will reveal a better performance
- reducing the cost of performing `UPDATE` and `DELETE` that implies cascading (`CASCADE`) or no action (`NO ACTION`)

As a rule of thumb, after schema modifications consider testing and monitoring of the indexes to ensure that current/additional indexes don't produce a negative

7. Add Columns for Index-Only Access

Adding columns for index-only access is a technique known as *index overloading*. Basically, we create an index containing all the columns needed to satisfy the query. This means that the query will not require data from the tablespace, therefore less I/O operations.

For example, consider the following query:

SQL

```
1 SELECT isbn
2   FROM book
3  WHERE genre = "History";
```

And the following index:

```
CREATE INDEX book_idx ON book (genre);
```

The index can be used for accessing columns with a given `genre`, but the database would need to access the data in the tablespace to return the `isbn`. By adding the `isbn` column to the index we have:

```
CREATE INDEX book_idx ON book (genre, isbn);
```

Now, all of the data needed for the above query exists in the index and no additional tablespace operations are needed.

8. Avoid Bad Standards

From coding style standards to recommended snippets of code for specific problems, companies love to use standards. Sometimes, among these standards, they sneak bad standards as well. One of the bad standards I saw says to limit the number of indexes per table to a certain value. This value varies between standards (e.g., 3, 5, 8), and this is the first sign that should raise your eyebrow that something is wrong here.

It doesn't matter how many indexes you have created per table! What matters is that every created index must increase or sustain the performance of your queries and doesn't cause significant issues in the efficiency of data modification. Data modifications (INSERT, UPDATE, DELETE) requires specific operations for maintaining the indexes as well. In a nutshell, database indexes speed the process of retrieval (SELECT) but slow down modification (INSERT, UPDATE, DELETE). So, as a rule of thumb, create as many indexes as are needed to support your database queries as long as you are satisfied by the trade-off between retrieval and data modification.

If you liked this article, then you'll my book containing 150+ performance items - **Spring Boot Persistence Best Practices** (<https://www.amazon.com/Spring-Boot-Persistence-Best-Practices-dp-1484256255/dp/1484256255/>).



Topics: JAVA, JPA, HIBERNATE, DATABASES, INDEX, PERFORMANCE

Opinions expressed by DZone contributors are their own.

Popular on DZone

- **The Unknown Design Pattern** (/articles/the-unknown-design-pattern?fromrel=true)
- **How to Implement OAuth2 Security in Microservices** (/articles/how-to-achieve-oauth2-security-in-microservices-di?fromrel=true)
- **What is LinkedHashMap? | LinkedHashMap Introduction and Sample Programs | Java Collections Framework** (/articles/what-is-linkedhashmap-linkedhashmap-introduction-a?fromrel=true)
- **Role of Continuous Monitoring in DevOps Pipeline** (/articles/role-of-continuous-monitoring-in-devops-pipeline?fromrel=true)

Database Partner Resources



Couchbase Cloud Public API Available!

The Public API opens new doors for automating operations on your Couchbase Cloud account. [Read the blog](#) ▶

Presented by **Couchbase**



2021 CI/CD Trend Report

Dive into the latest DevOps practices that are advancing continuous integration, continuous delivery, and release automation. [Download it](#)

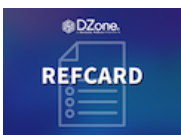
Presented by **DZone**



Get Started With Static Code Analysis

Explore the necessary steps for getting started with static code analysis, including CI/CD integrations, OWASP Benchmark, and more. [Read now](#)

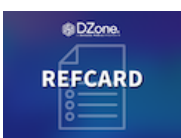
Presented by **ShiftLeft**



Multi-Region Database Deployments

Explore patterns & anti-patterns to multi-region database deployments — allowing your apps to survive a region failure without down

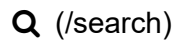
Presented by **Cockroach Labs**



Getting Started With Data Lakes

Learn how a data lake helps tackle data challenges through its enhanced architecture designed to ingest, store, and manage data. [Read now](#)

Presented by **Dremio**



REFCARDZ (/refcardz) RESEARCH (/research)
Careers (https://devada.com/careers/)

ZONES

+1 (919) 238-7100 (tel:+19192387100)

Visit the Writers' Zone ([/writers-zone](#))

Privacy Policy (/pages/privacy)

+1 (919) 678-0300 (tel:+19196780300)

(/pages/1660891111/psr/ufw/DZindex.do)rodDZcompny/dzone/)

(<https://devada.com/answerhub/>)