Indexing Beyond the Basics

A complete eBook about all database indexing fundamentals and secrets you should know

Tobias Petry

Table of Contents

Preface

Hey, welcome to *Indexing Beyond the Basics*! I am happy you decided it's time to learn everything you need about indexes.

Like everyone, I always wished I could solve the slow SQL queries I had written on my own. But even after reading some tutorials, I didn't get any closer to my goal.

So I learned everything about databases over the next 15+ years from countless books, conferences, trainings, articles and much practice. Finally, I could fix any slow query entirely without help. But during my consulting work, I found that most developers still face the same problem.

Unfortunately, I could never recommend good educational resources. All explanations are complex, difficult to understand and more tailored to database experts. They are just not helpful for a developer using databases.

So I worked for months on a completely new concept that you are now looking at: The entire book is enriched with dozens of illustrations that make it much easier to understand all concepts. Furthermore, the balance of practical and theoretical knowledge teaches you everything you need without being boring or daunting.

I hope that *Indexing Beyond the Basics* proves helpful! I am happy to hear your feedback.

1.2 Single vs Multi-Column Indexes

The biggest misconception about database indexes is that just creating an index on a column will make a query fast. It is simple, but not that simple...

And there is a big difference between creating an index on only a single column (which is most often done) and creating one with multiple columns.

So, let's build up a simple understanding of how an index helps the database to find the correct rows. For this explanation, you should imagine a customer service employee working for an insurance company searching for a specific contract: `WHERE firstname = 'Jane' AND lastname = 'Doe' AND insurance = 'health'`

Full Table Scan

The default query execution mode of a database is a table scan - also called a full table scan. As pictured in Fig. A, the database will load each row of the table one after another and check whether they match the query's condition. Matching ones will be kept while the other ones are discarded.

This is becoming slower with bigger tables as more rows have to be loaded and is the most inefficient approach for finding matching rows. However, it is nevertheless essential for the

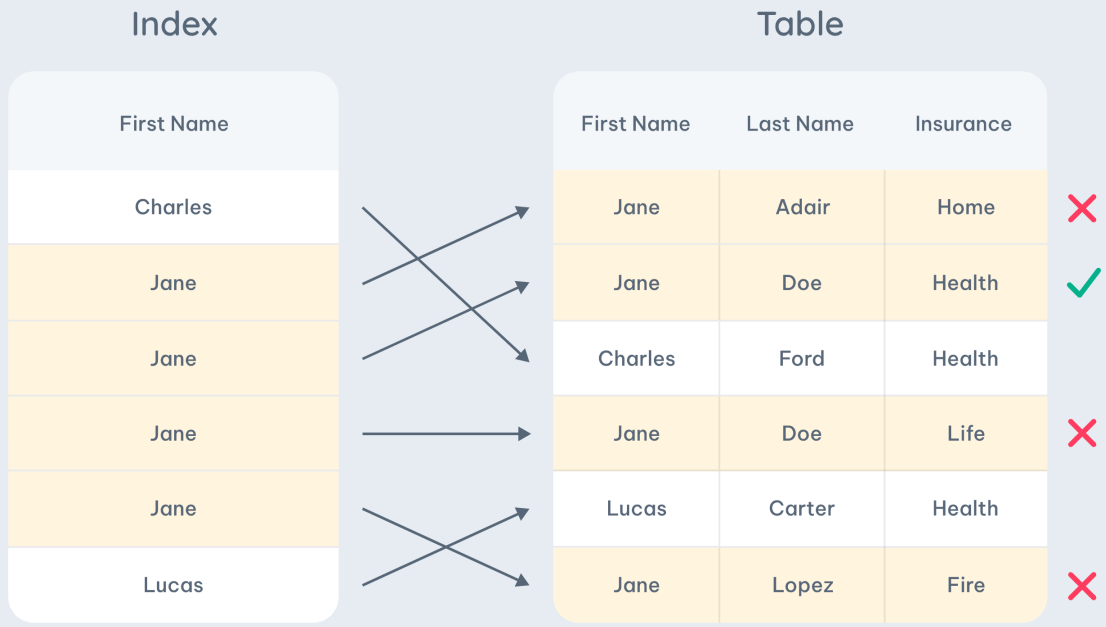
proper functionality of a database: You would not be able to execute some ad-hoc queries that you e.g. only run once interactively if a fitting index would always be required.

Single-Column Index

Therefore, creating an index is an optimization so the database can use a more efficient approach (if the indexes fits the query) and the response time for users is lower. The efficiency of queries often increases thousands of times as fewer rows need to be loaded from the disk. But as you can see, some rows are still loaded from the disk and discarded because they don't match the query's condition (Fig. B).

Figure B

SELECT With a **Single-Column Index**



The index records for Jane can be found and accessed easily (more on that in chapter "*2. Index Access Principles*") in the single-column index and all linked table rows are loaded. However, a single-column index can only narrow the rows to load based on the single column used. So, all conditions on other columns can only be checked after loading the row.

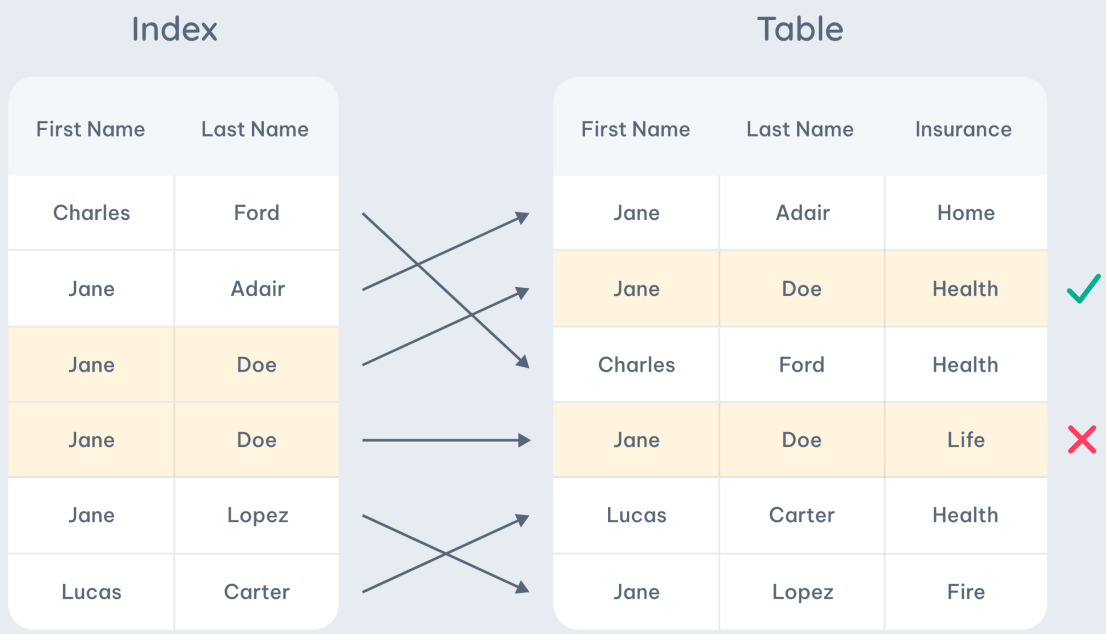
In this example, only three rows have been needlessly loaded. But a real insurance company will probably have tens of thousands of insured people named Jane. The query's runtime will be significantly impacted by loading too many rows, as the single-column index can't prefilter the table rows well enough.

Multi-Column Index

With the addition of another column to the index, it evolves into a multi-column index and the efficiency is increased. Now, two of the three conditions can be satisfied with the columns stored in the index. (Fig. C).

Figure C

SELECT With a **Multi-Column Index**



The database still loads too many rows as not all columns used in the query's condition are part of the index. However, the efficiency can be further improved by adding more columns to the index.

As the example shows, every query with more than one condition will be more efficient with a multi-column index than with a single-column one. Therefore, all the following chapters will discuss indexing based on multi-column indexes since they provide the most significant opportunity for performance optimization.

> **Performance**
>
> You should audit your database schema if you have many single-column indexes. Your queries most likely need more suitable indexes as countless rows are needlessly loaded and discarded - like in the example.

2. Index Access Principles

The most important aspect of creating good indexes is understanding how they are used by queries. And I don't mean to memorize and apply some examples. You have to build up an understanding of how a query can be mapped to an index. It is the only way to master index creation.

But nobody invented a rule system before to describe how to build good indexes. So, I created these four principles explained in this chapter that guide you on how any SQL operation can utilize an index.

1. Fast Lookup
2. Scan in One Direction
3. From Left To Right
4. Scan on Range Conditions

These principles teach you all the fundamental ways an index is used and the important concept of ordering columns in multi-column indexes. All following chapters build on this knowledge and the graphical illustrations showcasing the workflow of these principles.

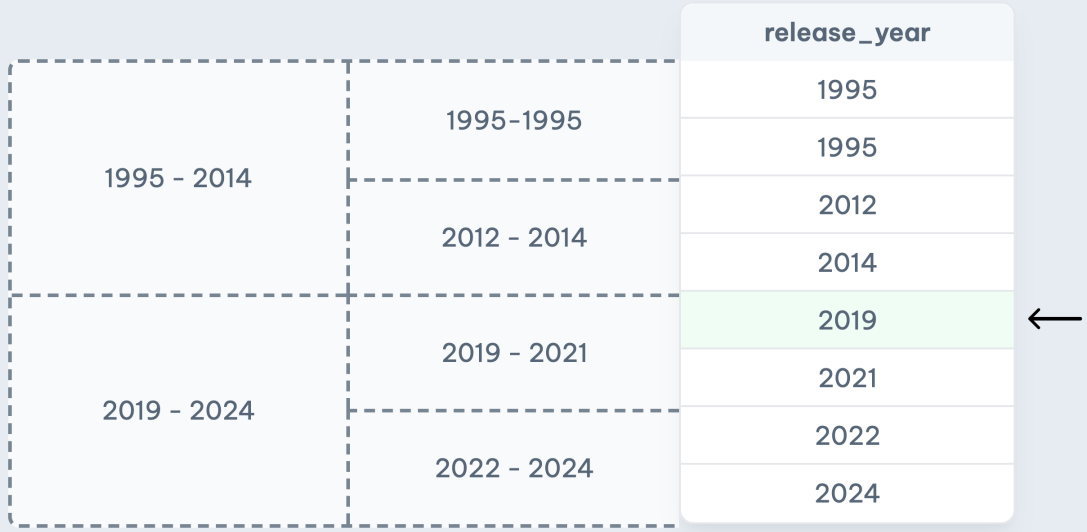
The objective is to learn a mental model to test whether a query can be mapped to an index! Drawing the visualizations shown here on paper is very helpful in the beginning. With more practice, you can do this entirely in your mind.

2.1 Principle 1: Fast Lookup

The most straightforward operation for an index is to find any value stored within it: You can expect it to jump almost directly at e.g. the offset for `WHERE release_year = 2019` within the sorted list without scanning it from start to finish by utilizing the index summary. This is visualized in Fig. A by the pointer right to the index entry that will indicate a fast lookup to a specific offset from now on.

Figure A

Index on **(release_year)**



It is still a common myth that a bigger index will result in slower queries. Indexes have been designed and optimized for this use case over the past decades! Searching within millions or billions of rows with an index is not slower than just a few thousand ones.

2.3 Principle 3: From Left To Right

An index on a single column is simple and easy to understand but the real problem and performance improvement opportunity is always with multi-column indexes - also called composite indexes. Once you have mastered using them, you can fix all slow queries yourself. Because of this, most of the rest of this book will focus on multi-column indexes.

The rule for multi-column indexes can be summarized as *"From Left to Right Without Skipping a Column"*. This simple sentence describes precisely how these indexes are used. However, it is hard to fully understand.

The fundamental constraint is that an index on the columns `firstname`, `lastname` and `country` (Fig. A) can only be used to speed up a certain type of query:

- `SELECT * FROM contacts WHERE firstname = 'James'`
- `SELECT * FROM contacts WHERE firstname = 'James' AND lastname = 'Walker'`
- `SELECT * FROM contacts WHERE firstname = 'James' AND lastname = 'Walker' AND country = 'US'`

Figure A

Index on **(firstname, lastname, country)**

Firstname	Lastname	Country
Aaron	White	US
Emma	Meyer	DE
Gerald	Babin	FR
James	Smith	US
James	Walker	GB
James	Walker	GB
James	Walker	US
James	Young	GB
Rosalie	Petit	FR

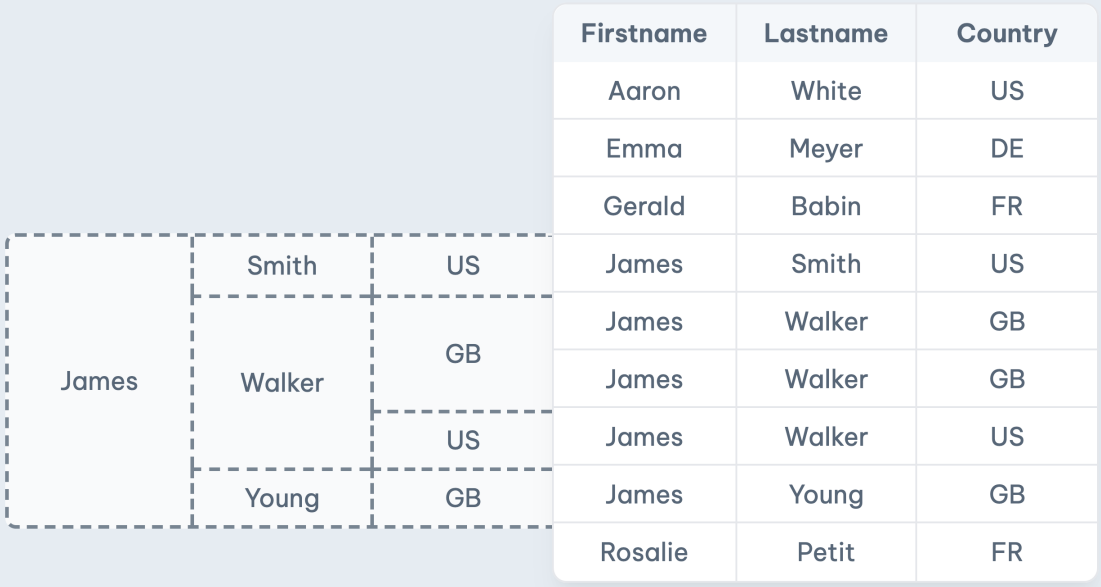
Indexes Are Used From Left to Right

It is important to remember that index entries are always sorted by the value of the first

column, all duplicate values are then by the second one and so on. The index summary can then be imagined as a funnel to narrow down the offset within the index by each column (Fig. B).

Figure B

Index on **(firstname, lastname, country)**



Firstname	Lastname	Country
Aaron	White	US
Emma	Meyer	DE
Gerald	Babin	FR
James	Smith	US
James	Walker	GB
James	Walker	GB
James	Walker	US
James	Young	GB
Rosalie	Petit	FR

The condition `firstname = 'James' AND lastname = 'Walker' AND country = 'US'` is executed like:

- Start at the top of the index by choosing the funnel for `firstname = 'James'`
- For the `lastname = 'Walker'` condition, choose now the middle row in the second column of the funnel. The row above for Smith and below for Young are now ignored as they can't have a result for the WHERE conditions.
- Go one step further to the right on the funnel and choose from the options `GB` and `US` the last one.

Finding the correct index entry was easy with the funnel-searching approach. But coming up with a funnel for doing this step so effortlessly sounds like cheating - it can't be that easy. In reality, the database is doing something exactly like that. The funnel imagination is just abstracting some B+ tree behavior from a complicated technical implementation to an easy-to-understand mental image. It is a simple way to decide whether a specific multi-column index would fit a query well.

The Ordering Is Important

You will find the incorrect advice repeated again and again that your multi-column index should start with the most selective (most different values) column first. The former index

has been rebuilt by this rule on the columns `lastname`, `firstname`, `country` (Fig. C). You can see that the number of funnel steps to get to the index entry is still the same - just ordering them by selectivity doesn't make a difference.

Figure C

Index on **(lastname, firstname, country)**

Lastname	Firstname	Country
Babin	Gerald	FR
Meyer	Emma	DE
Petit	Rosalie	FR
Smith	James	US
Walker	James	GB
Walker	James	GB
Walker	James	US
White	Aaron	US
Young	James	GB

But the ordering is still essential as an index has to fulfill your querying needs. If all your queries use all the index columns, the order won't make a difference (except for the next principle in the following chapter). But usually, most of your queries will use a subset of the indexes columns, while a tiny fraction will use all. In these cases, the ordering is critical!

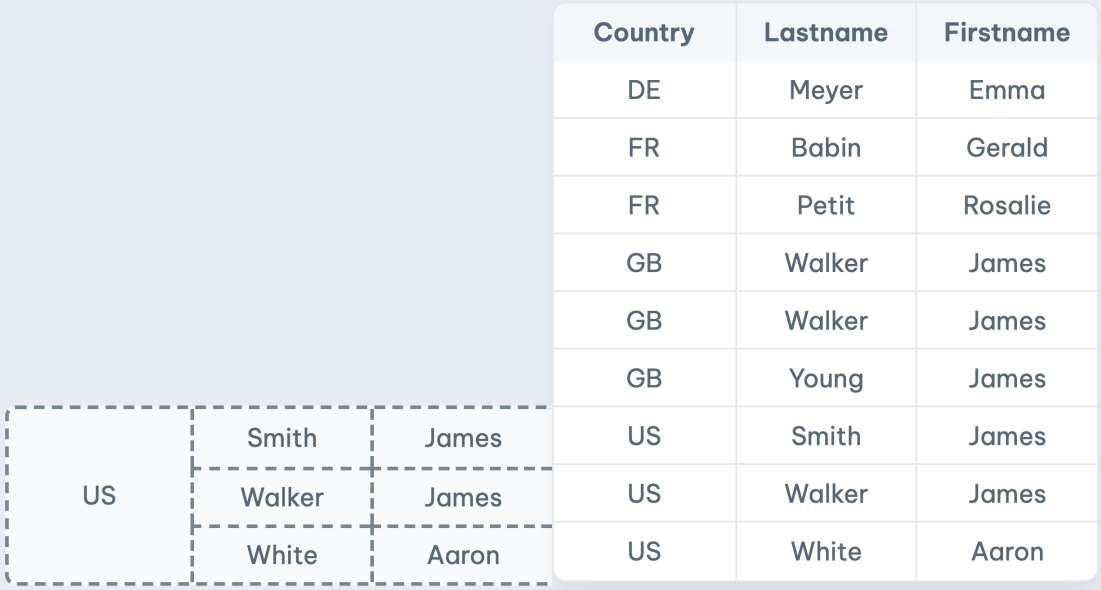
If you want to find all contacts from the United States (`WHERE country = 'US'`), neither the initial index on `firstname`, `lastname` and `country` (Fig. B) nor the one ordered by selectivity on `lastname`, `firstname`, `country` (Fig. C) can be used. Remember, an index is used from left to right because of the funneling approach. The `country` column would have to be the first one in the index to use the funnel - having them at the end does not help.

The correct approach to ordering index columns is to cover as many distinct queries as possible. The index of Fig. D on `country`, `lastname` and `firstname` is a perfect match if you execute these statements:

- `SELECT * FROM contacts WHERE country = 'US'`
- `SELECT * FROM contacts WHERE country = 'US' AND lastname = 'Walker'`
- `SELECT * FROM contacts WHERE country = 'US' AND lastname = 'Walker' AND firstname = 'James'`

Figure D

Index on **(country, lastname, firstname)**



There is no generic rule to follow for the ordering of index columns. A multi-column index is always built with the left-to-right rule to fit the funnel approach for as many queries as possible - not only for the one you optimize currently.

Skipping a Column

There is another common misunderstanding in addition to the ordering myth explained before. It is believed that a condition like `firstname = 'James' AND country = 'US'` won't use the index `(firstname, lastname, country)`.

The database can use the `firstname` of the funnel but then can't continue with the next step as a condition on the last name is not used. With a multi-column index, the database can't skip any funnel steps to jump to the matching index entries (*Index Access Principle 1: Fast Lookup*)!

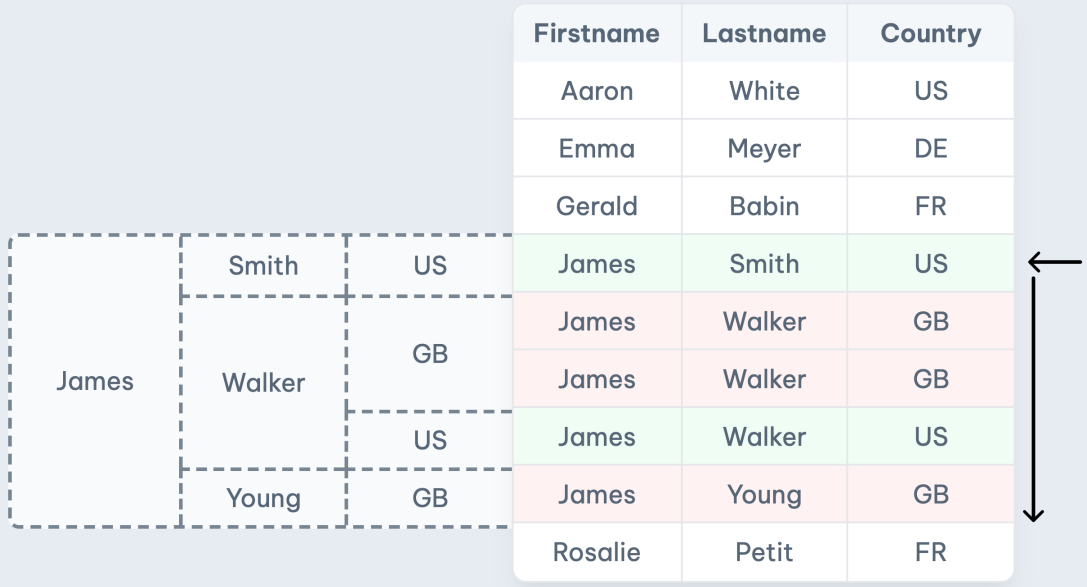
However, the index will still be used by the database but is less efficient than a perfect one on `(firstname, country, lastname)` or `(firstname, country)`. The steps involved are more complex (Fig. E):

- The database will use as many funnel steps as possible (without skipping a column) to narrow the potential index entries that could match the query. So in this example, there will be a fast lookup to the first index entry of James by using only the first column of the index (*Index Access Principle 1: Fast Lookup*).
- It will then proceed by iterating all index entries for James (*Index Access Principle 2: Scan in One Direction*).

- Each index entry will be validated whether they match the `country = 'US'` condition. Matching ones (green) are used, while non-matching ones (red) are discarded.

Figure E

Index on **(firstname, lastname, country)**



This procedure has to iterate over five index entries in this example and keep only two. While the perfect index would have been able to do a fast lookup on the first matching index entry and select both by scanning forwards. The difference may not be important for this small example. But your index may have hundreds of thousands of index entries for James in an actual application. Iterating over all of them to keep only the matching ones is much slower than directly finding the correct index entries when having a fitting funnel.

Although this approach is slower than the perfect index, it is still faster than a single-column index on `(firstname)`. Again, a single funnel step and iteration over the five index entries would be done. However, the iteration on the multi-column index can filter on the country column within the index and only load the two table rows that match the condition. The single-column index can't do this pre-filtering and has to load all five rows from the table to filter on the table's `country` column. The single column-index has to load more rows from the table which you want to avoid for best performance. Remember, for real applications, there could be thousands of rows for `James` but only a few would match the condition.

Overlapping Indexes

You may have overlapping indexes in your tables like the last index on `(country, lastname, firstname)` and e.g. a single-column index on `(country)`. Both indexes can be used to search for all contacts from the United States, while the multi-column index can

be used for more queries to further narrow down that selection. As indexes must be changed on every table modification, you should remove the single-column index in this example. You don't get any benefits by keeping it - the multi-column index speeds up the same queries.

This advice is also valid when comparing two multi-column indexes when their ordering is the same but one has more columns included than the other, e.g. `(country, lastname)` can be removed in favor of `(country, lastname, firstname)`. But based on the left-to-right rule, the indexes on `(country, lastname, telephone)` and `(country, lastname, email)` are different because one is not a strict superset of the other.

> **Performance**
>
> "From left to right without skipping a column" is a sentence that should be burned into your brain. While the first part of the rule is a requirement, the second one is optional. But for acceptable performance, both of them should always be satisfied!

3.6 Joins

Optimizing joins is a fascinating topic. When looking first at them, none of the things you learned before appear to match this complicated thing. How should the following query be optimized? This is using multiple tables and not just one!

```
SELECT employee.*
FROM employee
JOIN department USING(department_id)
WHERE employee.salary > 100000 AND department.country = 'NR'
```

You must de-construct a query using joins to understand how it will be executed and which indexes are needed. The basic way databases execute a join is called a “nested-loop join”. It works precisely like a for-each or for-in loop in any programming language: One table is accessed with all the filters applied (e.g. the `employee` table), and the matching rows will be the iteration data for the loop. For every one of these rows, logic similar to a new query on the `department` table will be executed by filling in the values from the `employee` table. You can imagine a join as highly efficient queries being executed within loops.

```
SELECT *
FROM employee
WHERE salary > 100000;

-- for each matching row from employee:
SELECT *
FROM department
WHERE country = 'NR' AND department_id = :value_from_employee_table;
```

The join optimization approach is now more manageable by having two independent queries. All the existing knowledge can be used to create the indexes on the employee (Fig. A) and department (Fig. B) table. For this example, the column order of Fig. B doesn't matter as both are equality checks and there are no other queries to optimize for (*Index Access Principle 3: From Left To Right*).

Figure A

Index on **(salary)**

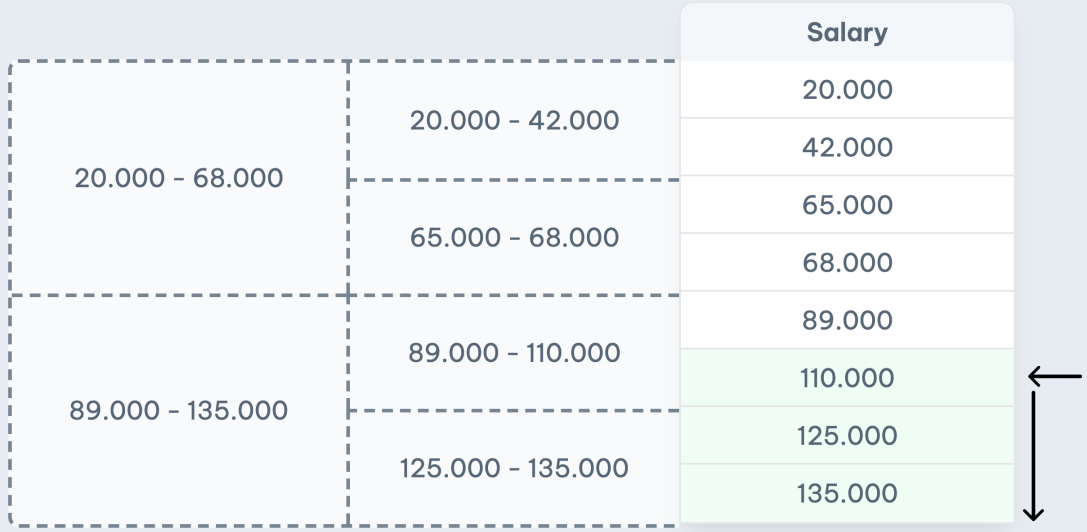
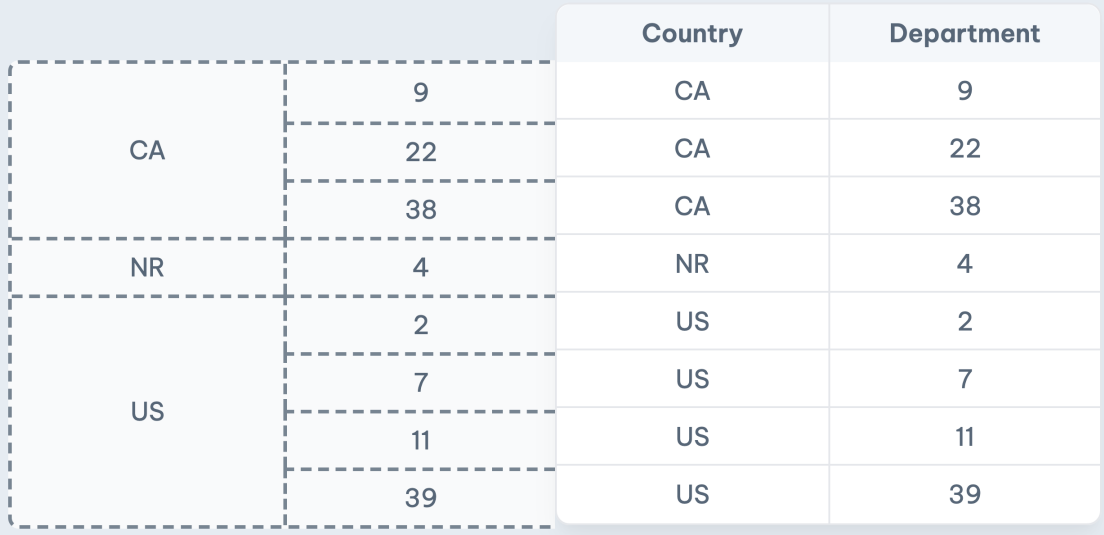


Figure B

Index on **(country, department_id)**



The approach for a two-table join can also be applied to queries involving joins with many more tables. The database will just do more nested loops than the one used in this simple example.

The Join-Order Is Not Fixed

SQL is a declarative language that specifies what data you want, e.g., how tables are linked together, how the rows should be sorted, and more. But you are not telling the database how to do this. How to execute each query is up to the database optimizer to find the fastest approach to retrieve your result.

Figure C

Distribution of Values

Country	Departments	Employees >100k
Canada (CA)	6	98
Nauru (NR)	2	2
United States (US)	24	411

We see an interesting case when looking at the aggregated example data for the query (Fig. C): The company only has departments in North America (Canada and United States) except for the small island of Nauru. Our approach filtered first on the employee table by narrowing it down to the 511 employees earning more than $100k/year. For each one, the department table was checked to only keep employees working for a department in Nauru. But couldn't the query be faster considering that Nauru has only 2 departments in total?

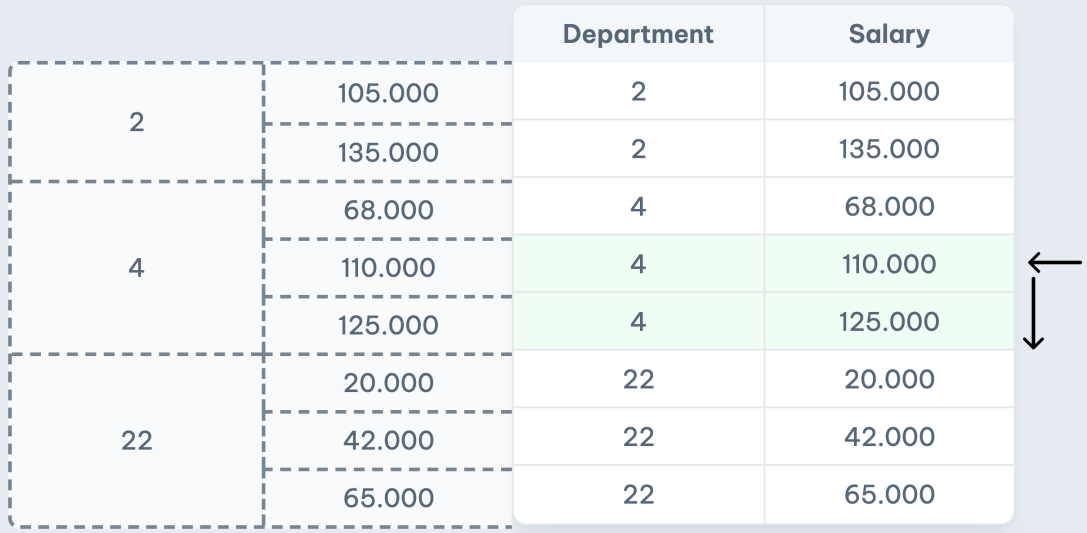
```
SELECT *
FROM department
WHERE country = 'NR';

-- for each matching row from department:
SELECT *
FROM employee
WHERE salary > 100000 AND department_id = :value_from_department_table;
```

The number of operations needed is reduced by switching the join order: First, the two departments of Nauru are found. Then, the employee table is searched with the new index of Fig. D for people earning more than $100k/year in each department. Only two queries are

executed within the loop compared to 511 ones.

Figure D

Index on **(department_id, salary)**



The order in which you write joined tables is not the order in which they are executed! As shown, a different execution order can make a query much faster and the database will try to estimate the fastest approach. But the correct indexes need to exist to let the database make this choice. Therefore, you should always add all indexes to execute joins in any possible ordering. If you omit an essential index, the database may never use the fastest join order.