Indexing Beyond the Basics

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

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.

Why You Didn't Understand Indexes Until Now

The reason database indexes are still not understood by developers is the depth of the existing content. Whether it is books, articles or videos - they all have their own problems that limit the ability to understand database indexing.

Blog articles and YouTube videos explain indexes by short examples on the most trivial problems (e.g. a missing single-column index) and won't go in-depth any further. But you cannot learn the broad topic of indexes from small examples. After all, the minimal knowledge taught is not enough to apply it to your more complex queries.

More knowledge is communicated in published books. However, they challenge everyone in another way: They are written for database administrators (DBA) and teach everything from backup approaches over complex database internals to obscure tuning settings that make it hard for developers to get actionable advice. Most likely, you have already stopped reading at least one of these books with 500+ pages because they include too many topics you are not interested in. In the end, database indexes were only covered on a few pages and were not sufficiently explained to fix your complex queries because the target audience of these books is interested in other topics.

These resources are based on either practice or theory. But both approaches must be combined for database indexes: You cannot learn theory from examples. And it is challenging to learn the practical application possibilities from theory. Therefore, theory and practice will alternate in the following chapters and you will only learn necessary theory that is needed later on. This book is exclusively about everything you must know about database indexes as a developer without any unnecessary topics!

Indexes are a complex subject because you need to understand all the involved concepts to master them. Therefore, the book should be read from cover to cover the first time as the later chapters build on the knowledge of the previous ones. After that, you can use each chapter independently as a manual to look up specific topics because they are written to be self-contained. All references to previous or later chapters only exist to read up on needed foundational knowledge again or point out some problems and optimizations you should know of.

I also recommend reading the book several times with some time in between. As your knowledge increases, you will discover new things you have missed before.

1. Fundamentals

I understand you want to jump in and create great indexes as fast as possible. But you'll have to be patient for a very short moment. Because if you focus purely on the practical stuff, you will miss critical foundational knowledge.

Databases are very complex technologies that consist of hundreds of thousands to millions of lines of code. A small amount of theory is necessary even though this book is focused only on practical topics.

I beg you not to skip this part. It is intentionally kept short - only a few pages. That said, understanding the basic idea of B+ trees and how they work together with tables is essential. And you will also learn an important lesson about schema design in this chapter.

1.1 A Different View on B+ Trees

B+ trees are the data structure why indexes are so fast. Every other book would start by explaining all the technical aspects: The distinction of leaf nodes (the most bottom rectangles) vs. inner nodes (the one on top and all in the middle), the different algorithms to insert and delete new values, rebalance the tree and much more.

You can read all the details about them on Wikipedia. But I won't repeat it for a specific reason: Do you want to know how often that knowledge is really needed? Honestly? This is just theoretical knowledge that will not take you any further. The number of people needing to know these technical details is tiny. And you are not one of them.

As a matter of fact, it will even complicate your understanding of indexes. You must read lengthy explanations and have difficulty understanding them because those damn trees can be complicated. As you read further chapters, you will never forget that you probably didn't fully comprehend a fundamental topic and never get the feeling of understanding everything. This is not a good learning experience.

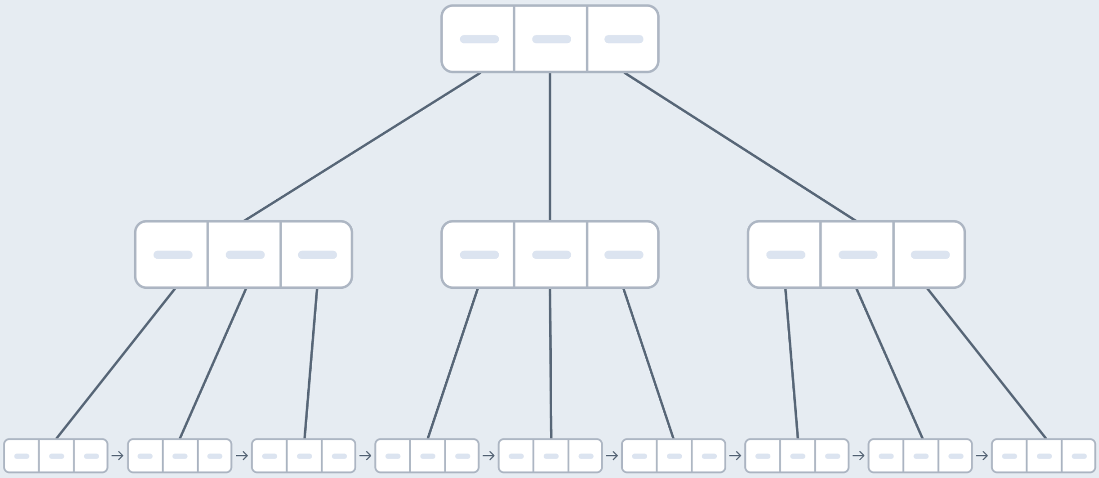


Figure A

An Exemplary B+ Tree

A More Understandable Approach

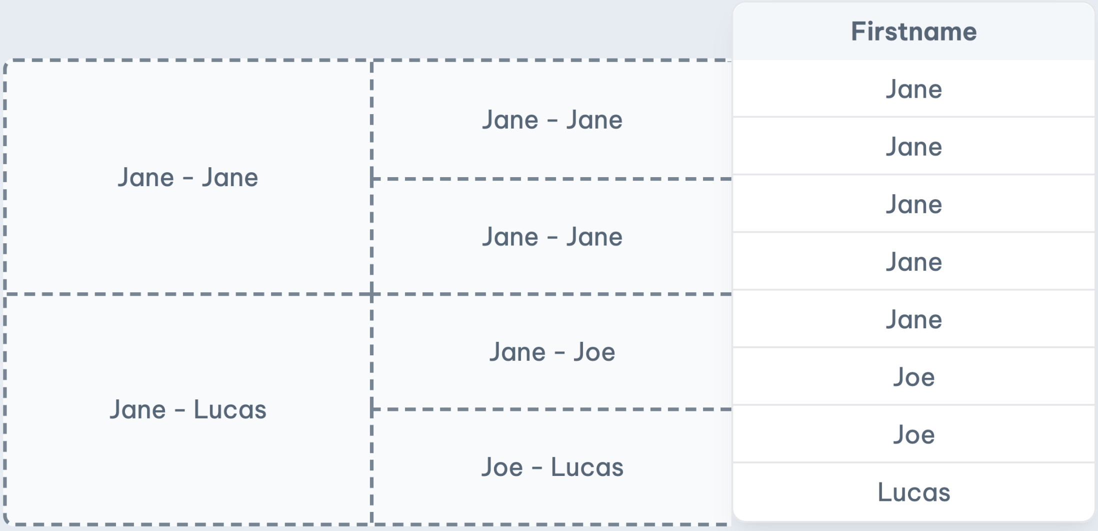
From now on, the simplified index visualization of Fig. B will be used. It may look too simple, but B+ trees are not complicated if you focus on the principles they provide and not their technical implementation.

Their core idea is to provide:

- a sorted list of values (B+ tree leaf nodes) for all columns used in the index represented as the table
- multiple levels of index summaries (B+ tree internal nodes) as shown on the left

Figure B

A More Straightforward Visualization of an Index



Firstname
Jane
Jane
Jane
Jane
Jane
Joe
Joe
Lucas

Finding a specific value is simple using these hierarchical ranges of values. Even with millions of rows, only a few steps following the ranges to the index records are needed.

Another reason for not having to know the details is that the database manages the index fully automatically. You don't have to worry about anything. These steps are executed for every change to a table without you having to do anything:

- Row Added: A new index entry for the row's values is created.
- Row Removed: The index entry for the row is removed.
- Row Changed: The index entry is removed and a new one is added when any column used within the index is changed. No modifications are required for changes to unindexed columns.

So the index is automatically changed for every table modification of indexed columns. From this, you can conclude that every index has an impact. The more indexes you create, the slower write operations become as more indexes must be modified. But searching for any data without them would be slow. So you would guess it is always a balancing act on how many indexes must be created to keep the tradeoff between read and write queries within limits. But you don't have to think about this tradeoff in most cases because data is more read than written and tables only have a few indexes.

Sorted vs. Unsorted Values

The B+ tree has been optimized to add new entries anywhere within this sorted list. But adding them to the end is always faster and needs fewer operations. You will learn about this behavior in arguments that e.g. random UUID keys are slower than sorted values like UUIDv7, incremental numbers etc. But these discussions never specify the databases they are for - this makes a big difference. In the next chapter (“The Interaction of Indexes and Tables”) you will learn that randomly ordered Primary Keys significantly impact performance with MySQL. For other databases, you can follow this advice for the best performance but also safely ignore this optimization if you don't have millions or billions of rows with many new ones inserted all the time.

1.2 The Interaction of Indexes and Tables

Knowing that indexes speed up queries is essential for fast performance. But you also need to understand how indexes, tables and loading of table rows work together. This cooperation opens up possibilities for optimization but is also the reason for queries continuing to be slow despite using an index.

The default operation to execute queries is doing a table scan - also called a full table scan. The database will load each row one after another and check whether they match the query's conditions. Matching ones will be kept while the other ones are discarded. This is becoming slower the more rows a table has.

An index is a lookup structure linking specific index entries to table rows so the matching ones can be found directly compared to scanning the entire table. But how is this done?

First, all matching index entries for queries are calculated - as explained by the following chapters (*“1. Index Access Principles”* and *“2. Index Supported Operations”*). Then, all rows referenced by the index entries are loaded one by one from the table. Any condition that was not part of the index will now be evaluated on the loaded row to decide whether to keep or discard it. Therefore, a query can still be slow even when using indexes: An index can reduce the millions of records in a table to a few thousand that must be loaded. But another condition (not part of the indexes columns) further reduces the rows to just a few. Although an index was used, thousands of rows were loaded pointlessly from the table, even though only a few were needed.

> **Performance**
>
> Using an index is not a guarantee for fast queries. Conditions on non-indexed columns should not significantly reduce the number of selected rows by the index. For these cases, a better index should be created including these columns.

Furthermore, how tables are stored also has a significant impact on performance. They can be stored in different ways (Heap Tables vs. Clustered Index) with different performance characteristics. You can specify which approach a table should use to benefit from these differences in some databases like Microsoft SQL Server or Oracle. While with MySQL (InnoDB), every table uses the clustered index approach and PostgreSQL will always use heap tables.

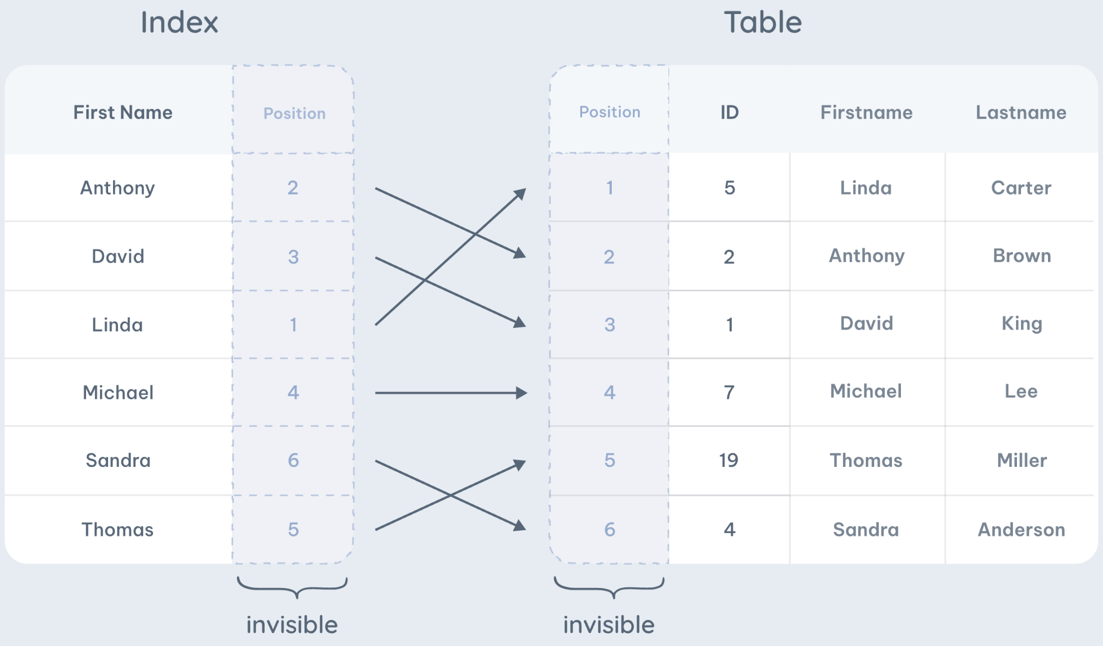
Heap Tables

Heap tables are the standard database approach because their implementation is simple. Any row inserted into a table is just appended to the end which is a fast operation because the end of the table will always be cached in the memory (because inserts are always done there). It does not matter if you insert the rows in the correct order of the primary key or not - they are always physically stored in the insertion order.

Index entries store the physical location of the row they refer to (Fig. A). So there is no difference between primary keys, indexes and unique indexes, as they all reference table row positions.

Figure A

The Index Points to the Physical Location of the Table’s Row



First Name	Position
Anthony	2
David	3
Linda	1
Michael	4
Sandra	6
Thomas	5

Position	ID	Firstname	Lastname
1	5	Linda	Carter
2	2	Anthony	Brown
3	1	David	King
4	7	Michael	Lee
5	19	Thomas	Miller
6	4	Sandra	Anderson

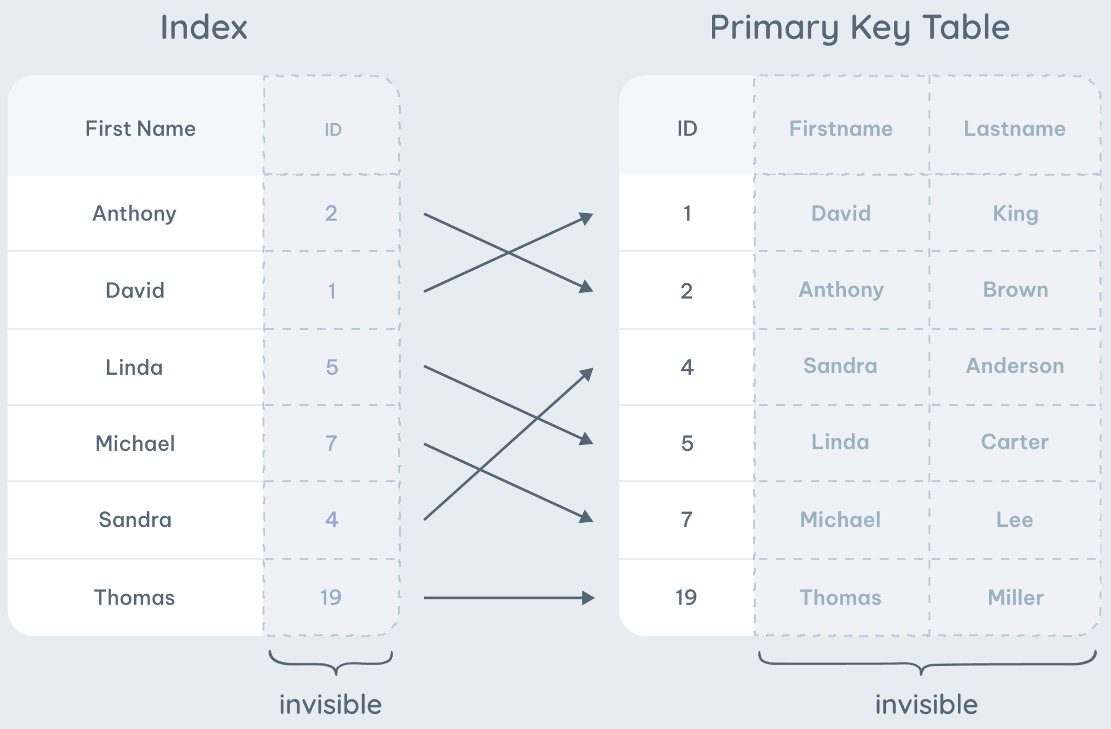
Clustered Index

There is no distinction between the primary key and the table when the database uses the clustered index storage method. They are the same: The table is eliminated by the primary key directly storing all the row's values. And indexes will not point to the table (as there isn't one) but instead reference the primary key by storing a copy of the primary key columns (Fig. B).

The most significant benefit of the clustered index is the increased primary key lookup performance: If rows are selected by their primary key (e.g. CRUD apps), all row values are directly available when the index entry is found. With the heap table approach, only the row's physical location in the table is known and the row still needs to be fetched. With secondary index lookups, there isn't a difference as both approaches need another step to load the identified rows.

Figure B

The Primary Key Is the Table



Index

First Name	ID
Anthony	2
David	1
Linda	5
Michael	7
Sandra	4
Thomas	19

Primary Key Table

ID	Firstname	Lastname
1	David	King
2	Anthony	Brown
4	Sandra	Anderson
5	Linda	Carter
7	Michael	Lee
19	Thomas	Miller

However, there are some pitfalls to be aware of. First, understanding the primary key behavior is critical when designing your schema. You shouldn't use a random value like UUIDv4 for your primary key because every new row must be inserted into a random position to guarantee the index order. The speed penalty of inserts to random offsets compared to just the end (with e.g. incremented integers) is massive. Don't do this!

You also need to take special care when choosing the primary key type: The database size will grow significantly if you use new identifiers like the 26-byte ULID string (if not using the smaller binary format) compared to a 4-byte incremented integer because the primary key columns are copied to every other index: A 22-byte bigger primary key for e.g. 4-5 indexes with a table of a million rows will increase the size by about 88-110MB. This may sound little but your server's memory is limited and you will have many more tables than just one. Therefore, the primary key type should be chosen considering its advantages and disadvantages. Just sticking to an incrementing integer is always a safe choice.

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.

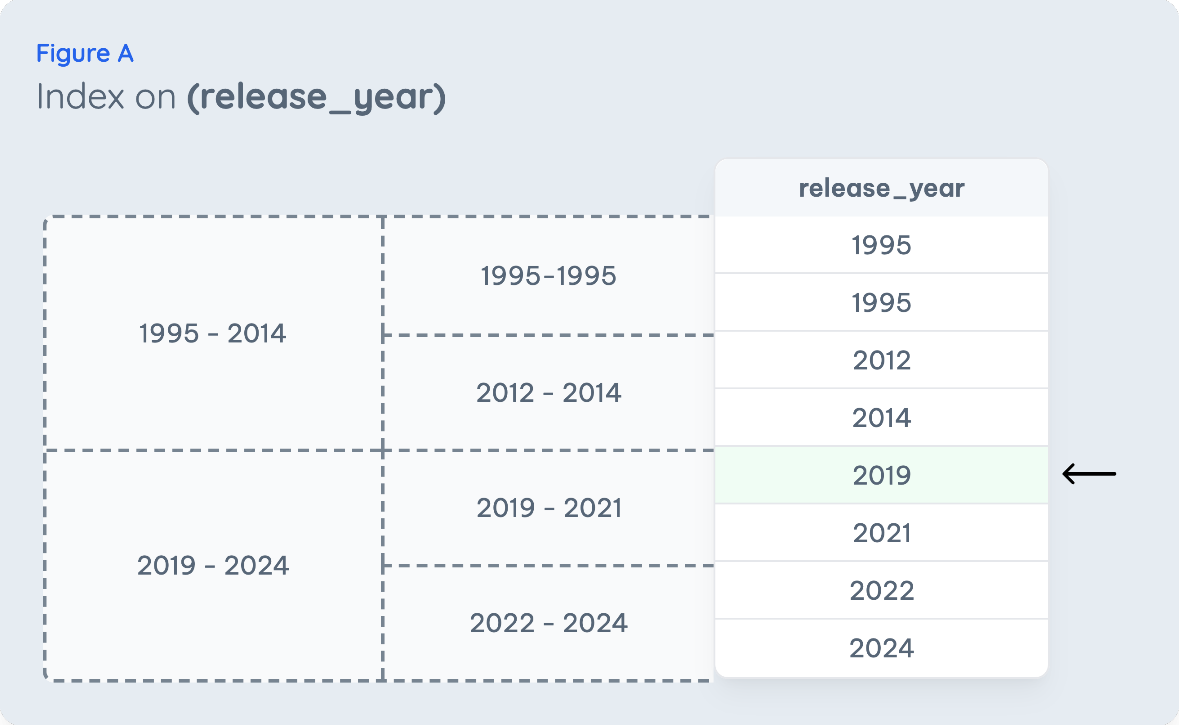
1. Fast Lookup
2. Scan in One Direction
3. From Left To Right
4. Scan on Range Conditions

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.



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.

2.2 Principle 2: Scan in One Direction

The second important rule of indexes is that they can do more than fast lookups. Whenever an offset within the index is found, the database can continue scanning the sorted list of values in either the ascending or descending direction. This is pictured in Fig. A, which executes `WHERE age >= 35 ORDER BY age ASC LIMIT 3` on the index:

1. The offset for the first index entry matching `age >= 35` is found.
2. The index entries are scanned in ascending direction
3. The process stops after the first three values are selected

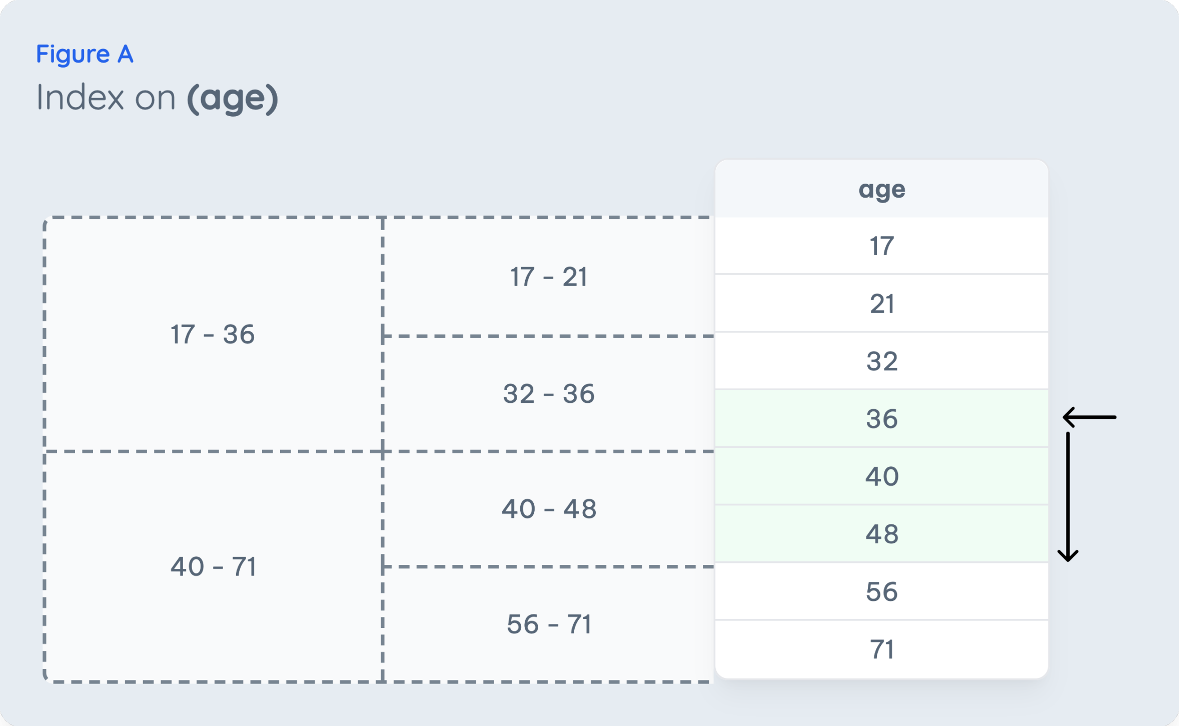


Figure A

Index on **(age)**

The same can be done by iterating the index values in the descending direction (`WHERE age <= 35 ORDER BY age DESC LIMIT 3`). But it is not possible to scan ascending and descending at the same time - it wouldn't make any sense either.

2.3 Principle 3: From Left To Right

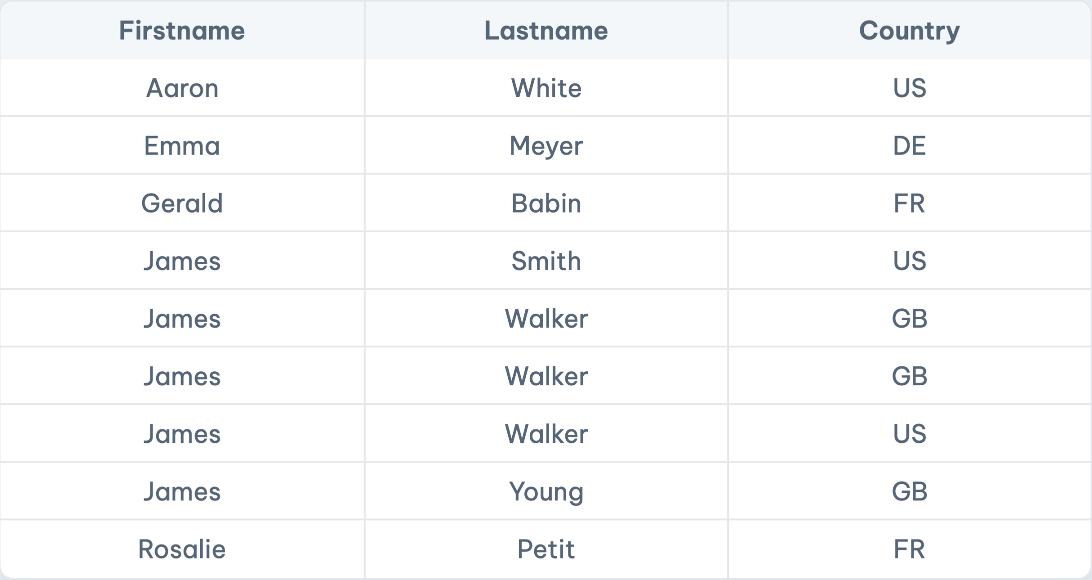
An index on a single column is simple and easy to understand but the real problem and performance improvement opportunity is always with multi-column indexes - also called composite indexes. Once you have mastered using them, you can fix all slow queries yourself. Because of this, most of the rest of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



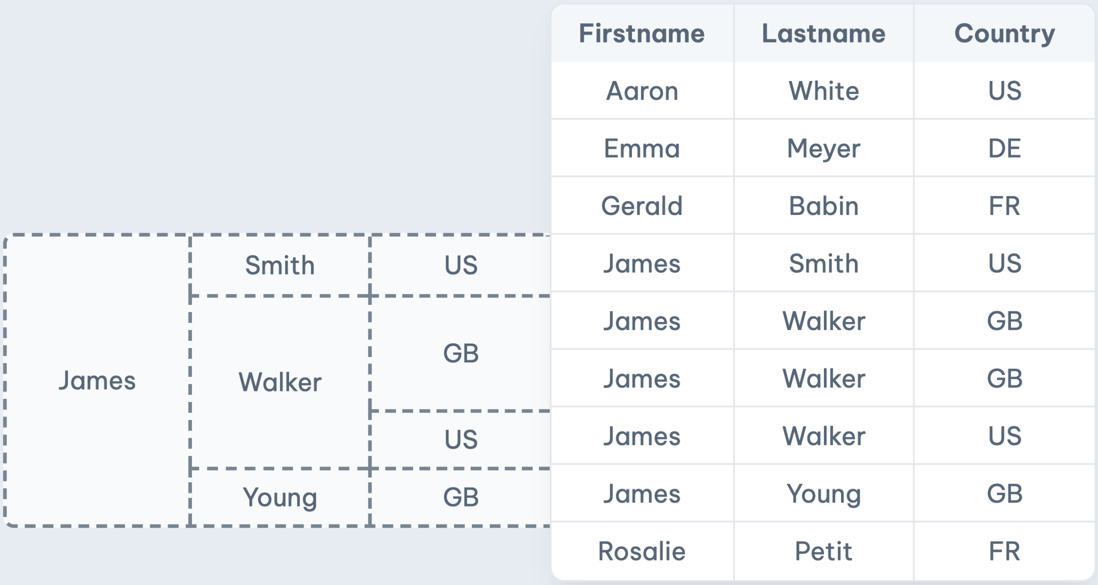
Firstname	Lastname	Country
Aaron	White	US
Emma	Meyer	DE
Gerald	Babin	FR
James	Smith	US
James	Walker	GB
James	Walker	GB
James	Walker	US
James	Young	GB
Rosalie	Petit	FR

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.

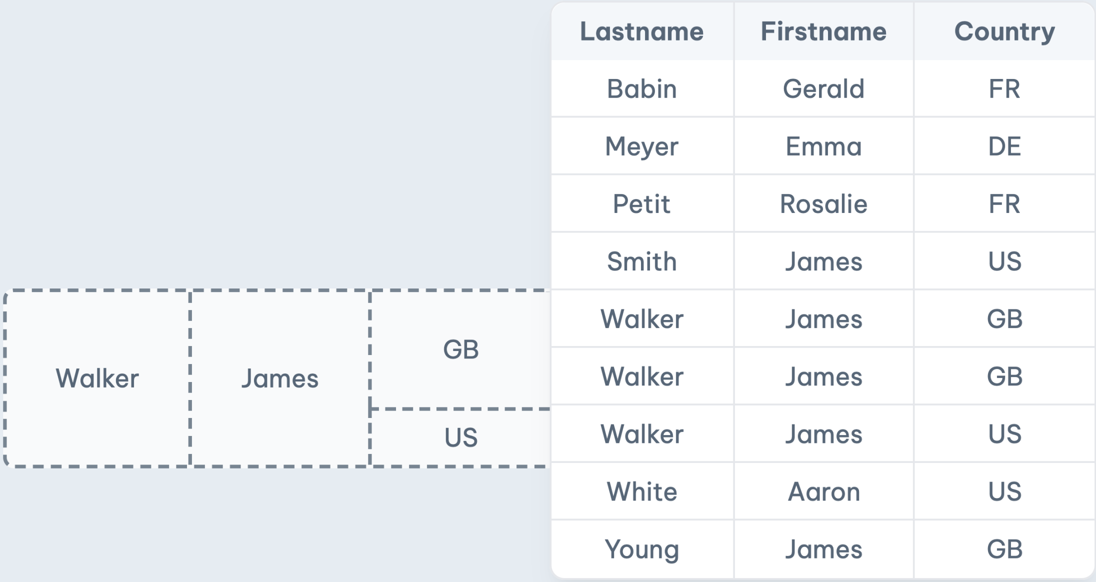
Finding the correct index entry was easy with the funnel-searching approach. But coming up with a funnel for doing this step so effortlessly sounds like cheating - it can't be that easy. In reality, the database is doing something exactly like that. The funnel imagination is just abstracting some B+ 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!

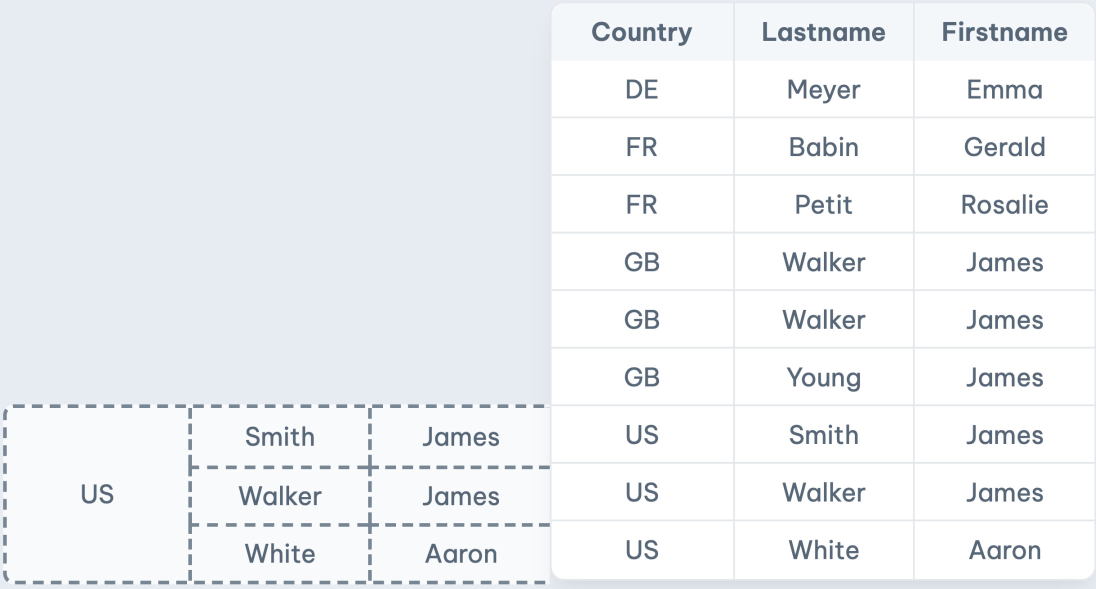
If you want to find all contacts from the United States (`WHERE country = 'US'`), neither the initial index on `firstname`, `lastname` and `country` (Fig. B) nor the one ordered by selectivity on `lastname`, `firstname`, `country` (Fig. C) can be used. Remember, an index is used from left to right because 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)**



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

US	Smith	James
	Walker	James
	White	Aaron

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*)!

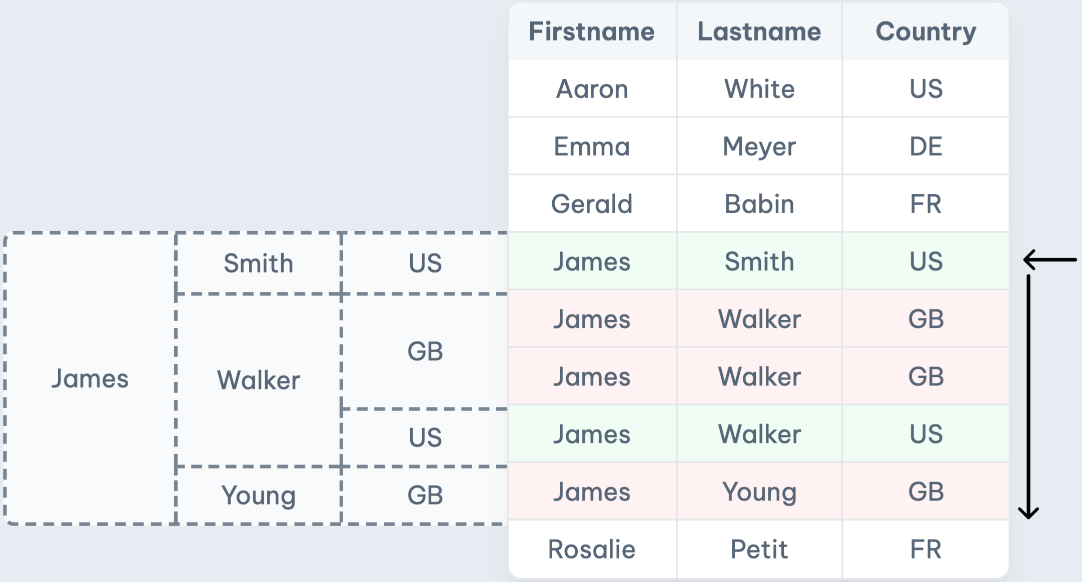
However, the index will still be used by the database but is less efficient than a perfect one on `(firstname, country, lastname)` or `(firstname, country)`. The steps involved are more complex (Fig. E):

- 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)**



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

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!

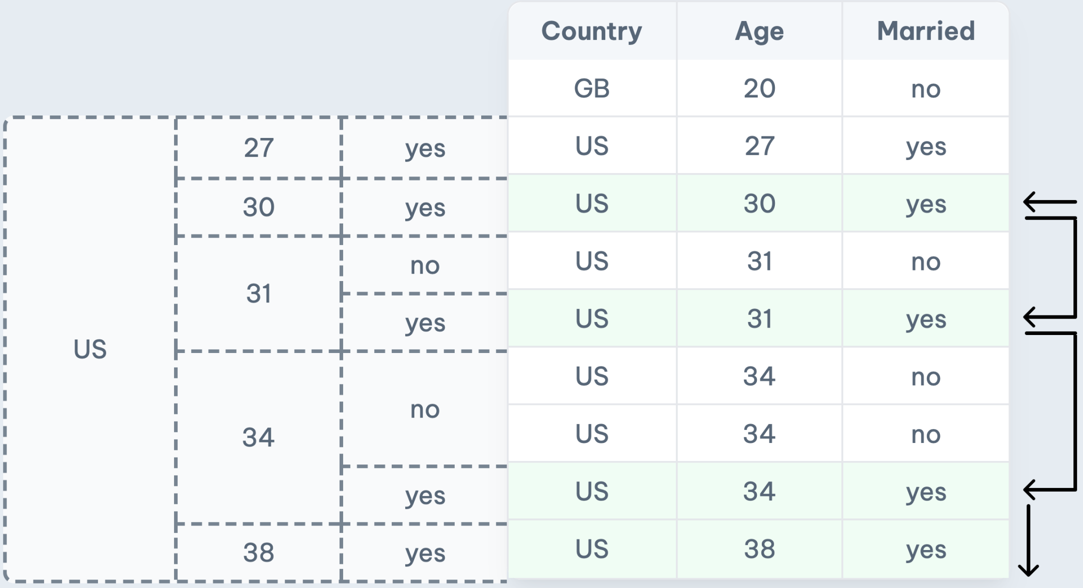
2.4 Principle 4: Scan On Range Conditions

There is one last rule for multi-column indexes you need to understand. All examples before have been using equality checks, but the behavior slightly changes when you use a range condition like `<` `<=` `>=` `>`.

How do you believe a condition like `WHERE country = 'US' AND married = 'yes' AND age > 28` is executed for an index on `(country, age, married)`? You may expect a fast lookup for the first index entry, a scan in ascending direction and skipping non-matching index entries by using the last step of the funnel - as pictured in Fig. A.

Figure A

Index on **(country, age, married)**



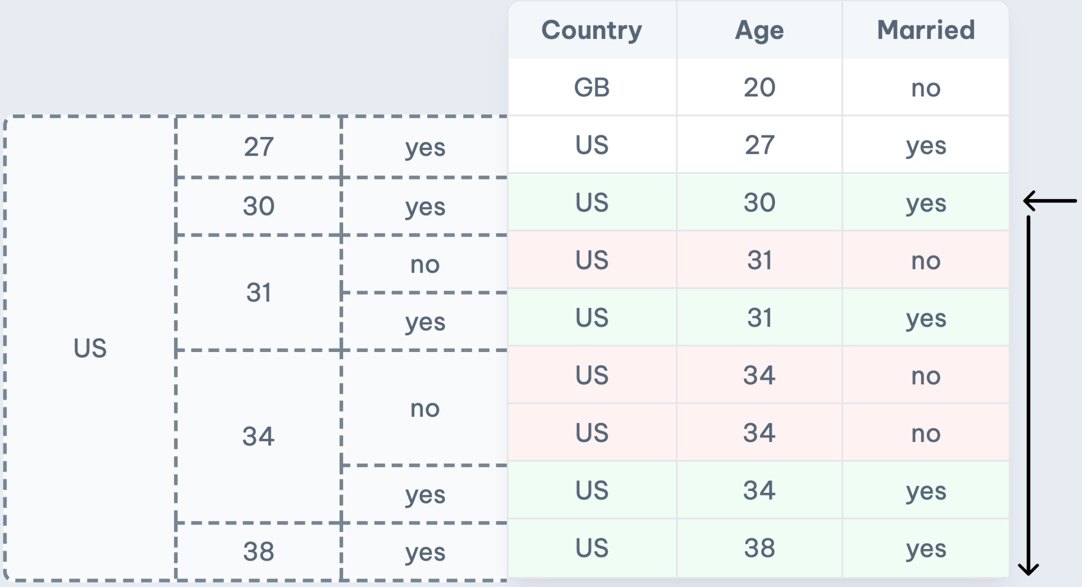
Country	Age	Married
GB	20	no
US	27	yes
US	30	yes
US	31	no
US	31	yes
US	34	no
US	34	no
US	34	yes
US	38	yes

But the behavior is different: When starting a scan in one direction, the database can't skip any index entries anymore. Any range condition will always start the scanning process and the number of index entries to touch can't be reduced anymore! The workflow will look as pictured in Fig. B by doing a fast lookup and scanning all matching index entries for the condition on the `age` column. As in the last chapter, the `married` column is still used to filter index entries and load only the matching rows from the table, but it can't limit the number of index entries scanned.

The ordering must always be adjusted so that columns with an equality check are used before a range condition to reduce the number of scanned index entries (Fig. C).

Figure B

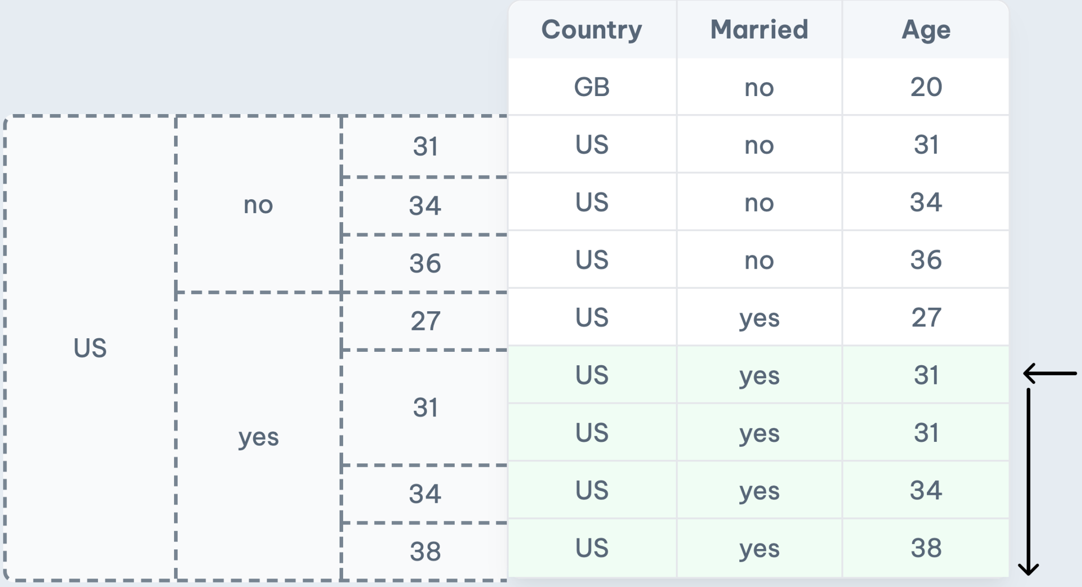
Index on **(country, age, married)**



Country	Age	Married
GB	20	no
US	27	yes
US	30	yes
US	31	no
US	31	yes
US	34	no
US	34	no
US	34	yes
US	38	yes

Figure C

Index on **(country, married, age)**



Country	Married	Age
GB	no	20
US	no	31
US	no	34
US	no	36
US	yes	27
US	yes	31
US	yes	31
US	yes	34
US	yes	38

The statement that index entries cannot be skipped is not always valid. Some databases can do this for spcific `GROUP BY` queries that want to get the minimum or maximum value for each group. MySQL calls this “Loose Index Scan” and Microsoft SQL Server and Oracle use the term “Skip Scan” to describe this optimization. However, it is not available with all databases - e.g. PostgreSQL.

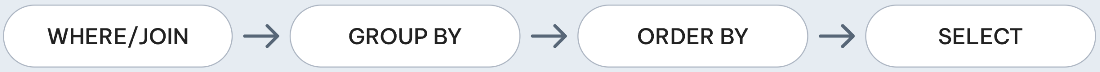
3. Index Supported Operations

The index access principles explained before are everything you need to create perfect indexes for any query. By following those guidelines, you can spot whether e.g. the column order for a multi-column index violates some principles and needs to be changed. But your queries are not only using a few WHERE conditions with equality checks. Let's expand these principles to see how they work on complex queries and operations.

The SQL execution order (Fig. A) is essential when dealing with more complex queries. Before rows are sorted (ORDER BY), they must be processed by all former steps (WHERE/JOIN and GROUP BY) - if used by the query. An index must consequently always satisfy all prior execution steps to be used for a specific step: A query using filtering and ordering must have the filtering columns in the index before the sorting ones. But more on that in each specific chapter.

Figure A

SQL Execution Order



Info

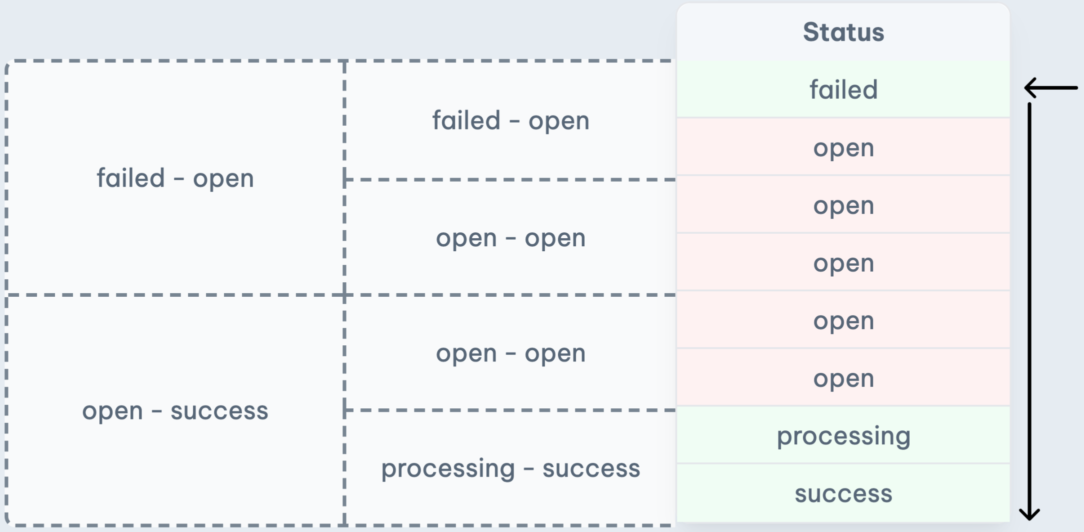
This chapter follows the introduction of the index access principles to continue the learned theory with many practical examples. Some comments that an index may not be used may appear weird to you. If you stumble upon such a passage, reread it after finishing the chapter “*4. Why Isn’t the Database Using My Index?*”.

3.1 Inequality (!=)

Inequality conditions can quickly kill your performance. A condition to e.g. search for all payments not in a specific state (`WHERE status != 'open'`) is quite complicated to execute. This condition can only be mapped to an index by iterating all index entries and checking each for a matching `status` value (Fig. A). Remember, a scan operation can not skip index entries so the entire index needs to be scanned.

Figure A

Index on **(status)**



Unfortunately, the database can not predict how many index entries would match the condition and consequently falls back to scanning the entire table (not the index!) to satisfy the query. You will learn more about this behavior in chapter *4. “Why Isn’t the Database Using My Index?”*.

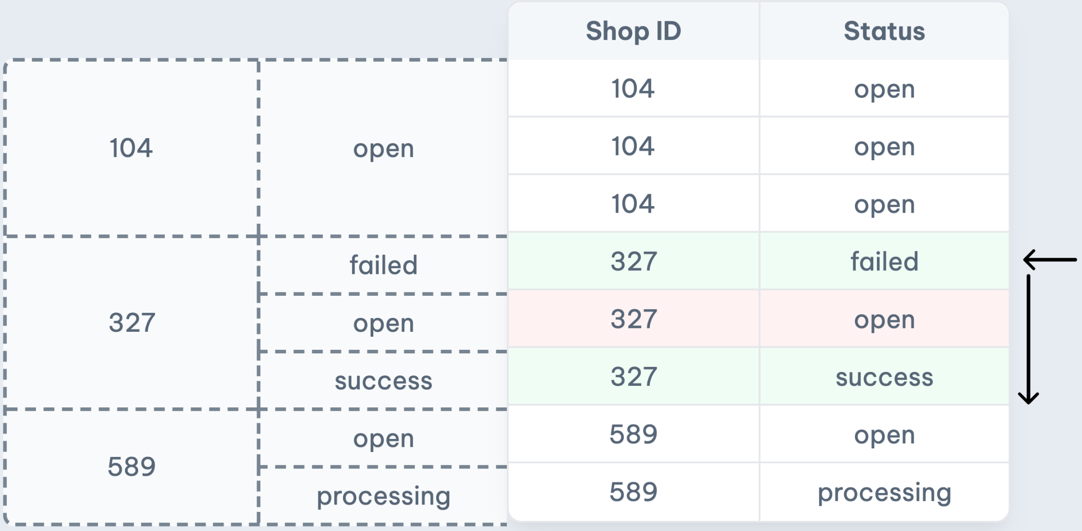
So inequality conditions are problematic for tables with more than a few hundred rows and impossible to improve by just creating an index. You always have to make query changes to make those queries fast.

Combine With More Columns

An inequality condition is rarely used alone. The simplest form of optimization is to include another column of the query's condition in the index. For example, the query can be constrained to all payments for a specific shop. An index on both columns (Fig. B) will only scan all of this shop's index records. The database will no longer fallback to a table scan as it can predict an upper bound for the matching rows (all rows for the specific shop). Again, you will understand the reasons behind this after reading the formerly referenced chapter about the cost model. For now, you have to trust this explanation.

Figure B

Index on **(shop_id, status)**



Shop ID	Status
104	open
104	open
104	open
327	failed
327	open
327	success
589	open
589	processing

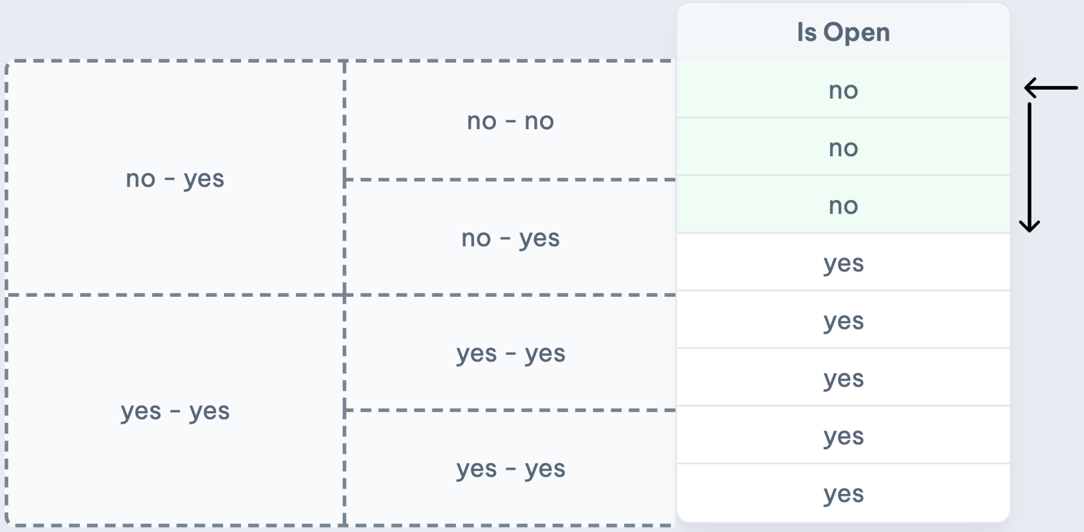
Transformation to an Equality Rule

The previous optimization made the query faster but it is still not optimal. Although only two index entries could match the condition, three entries had to be checked based on the query's `shop_id`. The ratio would have been much worse with more data and a different distribution of values in the table. Furthermore, adding additional conditions to a query is not always possible. So the previous optimization can not be used for all circumstances.

An excellent technique here is to convert the inequality condition to a boolean value: Either the condition matches the row (`yes`) or not (`no`). Therefore, a simple equality condition with a corresponding index must only scan the matching index entries (Fig. C). This optimization is further explained in chapter *5.3 Transforming Range Conditions*.

Figure C

Index on **(is_open)**



3.2 Nullable Values (IS NULL and IS NOT NULL)

The NULL value is an interesting edge case in SQL. It has a special meaning that describes the absence of a value - something that is unknown. Consequently, you have to handle this one differently: Following the SQL standard, an unknown (non-existent) value can't be compared to any other unknown (non-existent) value. Therefore, the equality check (`NULL = NULL`) and inequality check (`NULL != NULL`) will always be false. You must use the specialized conditions `IS NULL` and `IS NOT NULL` instead.

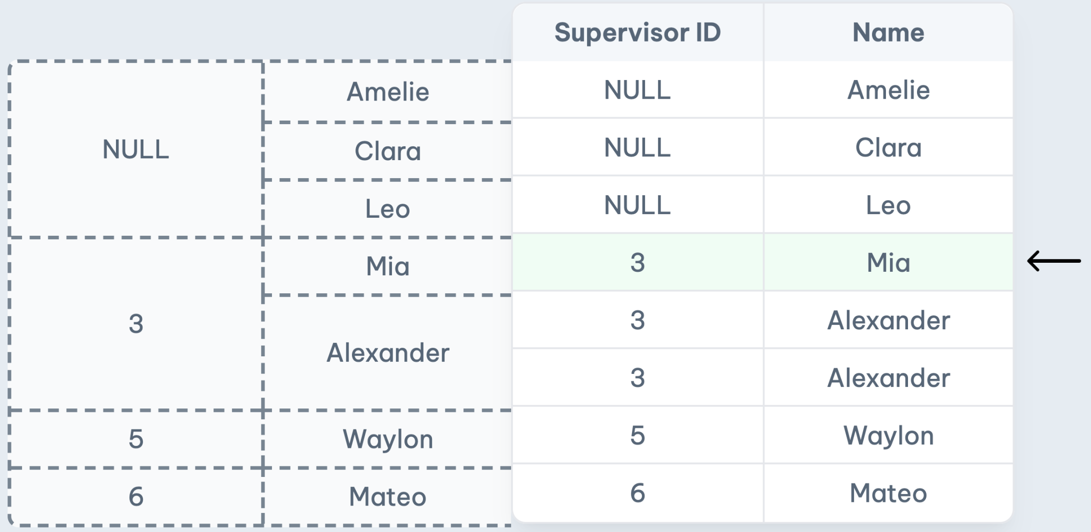
The SQL standard has never defined the exact order of whether NULL values are before or after other values. This decision is left to the database. For all our examples, NULL values will be placed before existing values.

IS NULL

The `IS NULL` condition works with the same semantics as an equality condition. Therefore, everything you learned before applies to `IS NULL` too. An index to search for the employee Mia who has no supervisor (`WHERE supervisor_id IS NULL AND name = 'Mia'`) can be created with the `supervisor_id` column first (Fig. A) or last (Fig. B). Both work the same and have the same efficiency. Which one to use depends on the other queries the application executes and which indexes can be reused most often by them (*Index Access Principle 3: From Left to Right*).

Figure A

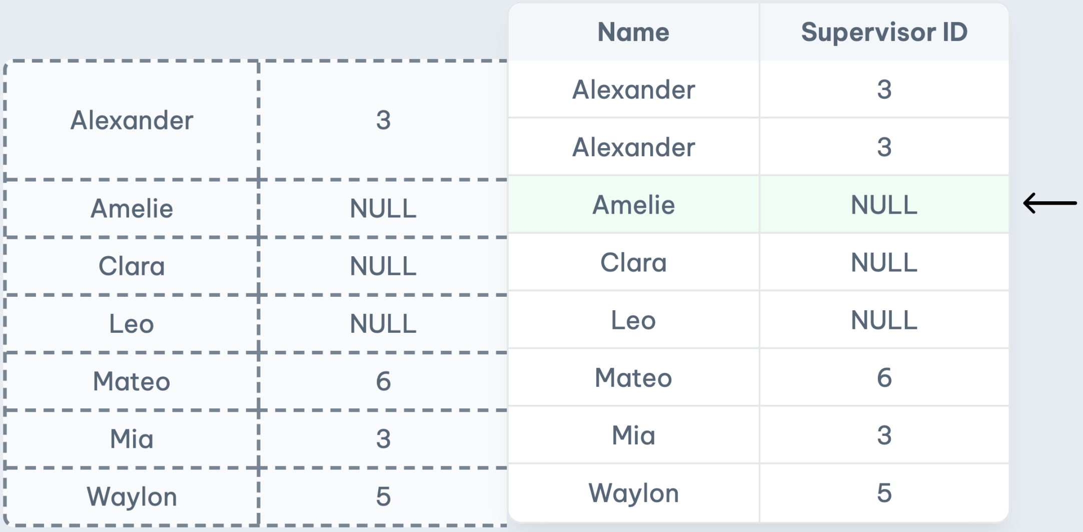
Index on **(supervisor_id, name)**



Supervisor ID	Name
NULL	Amelie
NULL	Clara
NULL	Leo
3	Mia
3	Alexander
3	Alexander
5	Waylon
6	Mateo

Figure B

Index on **(name, supervisor_id)**



Alexander	3
Amelie	NULL
Clara	NULL
Leo	NULL
Mateo	6
Mia	3
Waylon	5

Name	Supervisor ID
Alexander	3
Alexander	3
Amelie	NULL
Clara	NULL
Leo	NULL
Mateo	6
Mia	3
Waylon	5

IS NOT NULL

Again, the `IS NOT NULL` condition has the same semantics as the inequality condition, which also means that the index may not be used. Likewise, the same optimizations should be used.

3.3 Pattern Matching (LIKE)

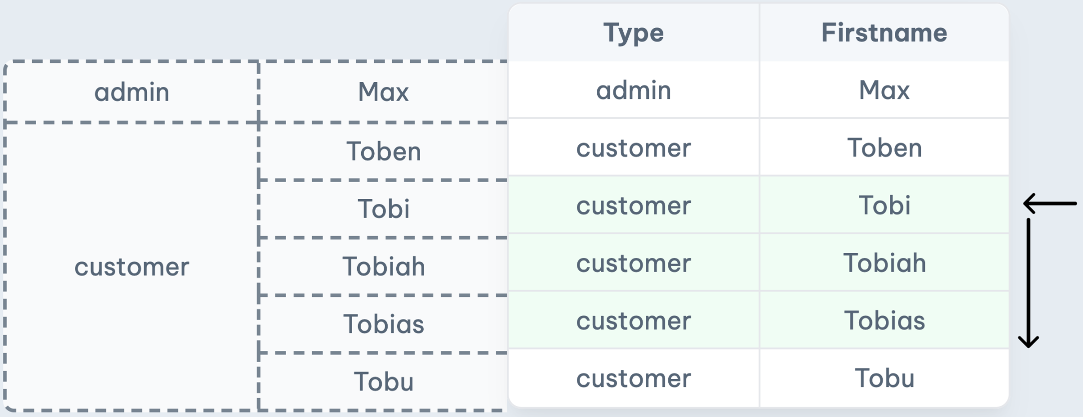
Searching with wildcards is used in many applications. Sometimes you search for a customer (`type = 'customer'`) and only know that their first name starts with Tobi (`firstname LIKE 'Tobi%'`). You are searching for rows matching a pattern and not a specific value.

A pattern matching condition is internally rewritten as a range condition (`firstname >= 'Tobi' AND firstname < 'Tobj'`) from the first possible value `Tobi` (exact match) to anything before `Tobj` (the first value that wouldn't match anymore).

So pattern-matching conditions start a scan in one direction with all their implied characteristics for query tuning (Index Access Principle 4: Scan On Range Conditions). Therefore, the column with the wildcard search should be after the equality column (Fig. A).

Figure A

Index on **(type, firstname)**



admin	Max
customer	Toben
	Tobi
	Tobiah
	Tobias
	Tobu

Type	Firstname
admin	Max
customer	Toben
customer	Tobi
customer	Tobiah
customer	Tobias
customer	Tobu

Pitfall

An index can only be used when the wildcard is placed in the middle or the end of the search string. The chapter "*5.4 Leading Wildcard Search*" explains why leading wildcards are not using an index.

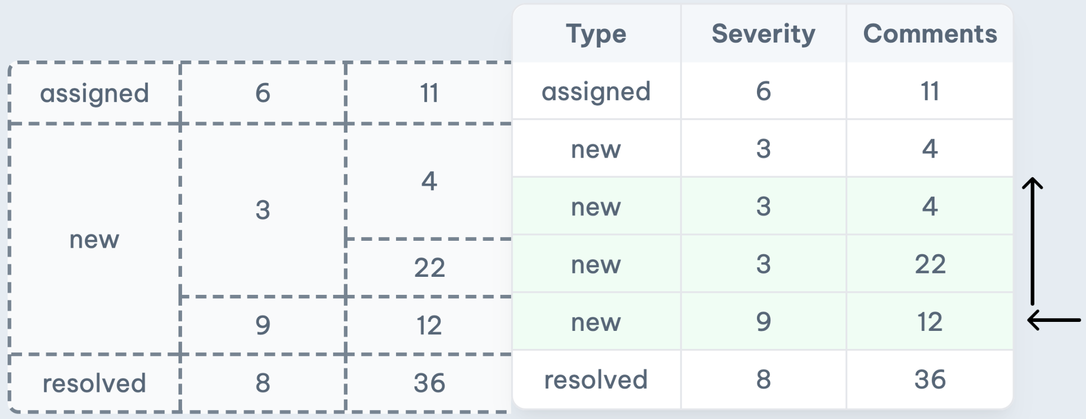
3.4 Sorting Values (ORDER BY)

Database indexes can be utilized for more operations than efficiently filtering WHERE conditions. If you add the sorting columns last to the index, you can also get index entries in sorted order so that no additional sorting step is needed.

A GitHub-like issue search functionality may filter by the type (`WHERE type = 'new'`) and sort to show the highest severity issues with the most comments first (`ORDER BY severity DESC, comments_num DESC`). As pictured in Fig. A, the first matching index entry can be found with a fast lookup for new issues and the highest value for both columns. From there, the index entries can be read by scanning in sorted order because all index values are pre-sorted according to the sorting condition.

Figure A

Index on **(type, severity, comments_num)**



Type	Severity	Comments
assigned	6	11
new	3	4
new	3	4
new	3	22
new	9	12
resolved	8	36

Avoid Additional Sorting Steps

An extra sorting step is needed when rows can not be read in the correct order directly from the index. This is reasonable when sorting a few rows, but you will experience severe performance problems sorting medium to big results.

Every database has a threshold for the maximum (temporary) query result size that can be kept in memory. If you exceed this limit, the data can no longer be sorted fast in the memory. A much slower approach using the disk is needed: All rows must be written to a file on the disk in small chunks to circumvent the limit of memory used at any time. Multiple chunks will then be read from the disk, sorted in the memory and written to the disk repeatedly until all rows are correctly ordered.

This complicated workflow clearly shows that you should absolutely avoid it. Remember, even with a `LIMIT` all rows must be sorted first to throw away the unneeded ones later. Always create an index to get rows sorted from the index when the conditions match many rows!

> **Expert Knowledge**
>
> With MySQL, you can increase the memory threshold for in-memory sorting with the `sort_buffer_size` configuration that defaults to 256KB to a more reasonable value. While PostgreSQL uses the generic `work_mem` setting that applies to any temporary in-memory results and is by default set to 4MB.

3.5 Aggregating Values (DISTINCT and GROUP BY)

Queries involving GROUP BY and DISTINCT are the most challenging problems when solving a performance problem. They are always left out when discussing indexes for some inexplicable reason. However, it is important to understand them since they often aggregate tens of thousands of rows. Minor issues like loading some information from tables rather than only using the index will slow down a query from a few milliseconds to many seconds.

Distinct

DISTINCT is the same as GROUP BY for query optimization. It is just syntactic sugar to make writing some queries easier as you must type less. The database will execute both following queries the same. So the rules you will learn for GROUP BY will also apply for DISTINCT when remembering that the query is internally executed like a GROUP BY query.

```sql
SELECT DISTINCT country FROM users;

SELECT country FROM users GROUP BY country;
```

GROUP BY

The essential idea of GROUP BY is to aggregate many rows by an aggregation function (e.g. count, avg, etc.) into one result for every group of similar columns. As many rows are involved in this operation, you want this to be as fast as possible and work entirely on an index's columns. Loading each involved row would result in a very slow query when aggregating hundreds of thousands of rows.

Time-intensive operations like sorting or temporary tables can be required if the query can not use an index or only for parts of the GROUP BY. Creating a suitable index is therefore absolutely necessary.

> **Tip**
>
> You must add all non-aggregated columns (e.g. `price` but not `AVG(price)`) within the SELECT part to the list of GROUP BY columns. But if you add the primary key of a table to the GROUP BY, you don't have to add any columns for that table anymore. The database will add all of them automatically in the background for you.

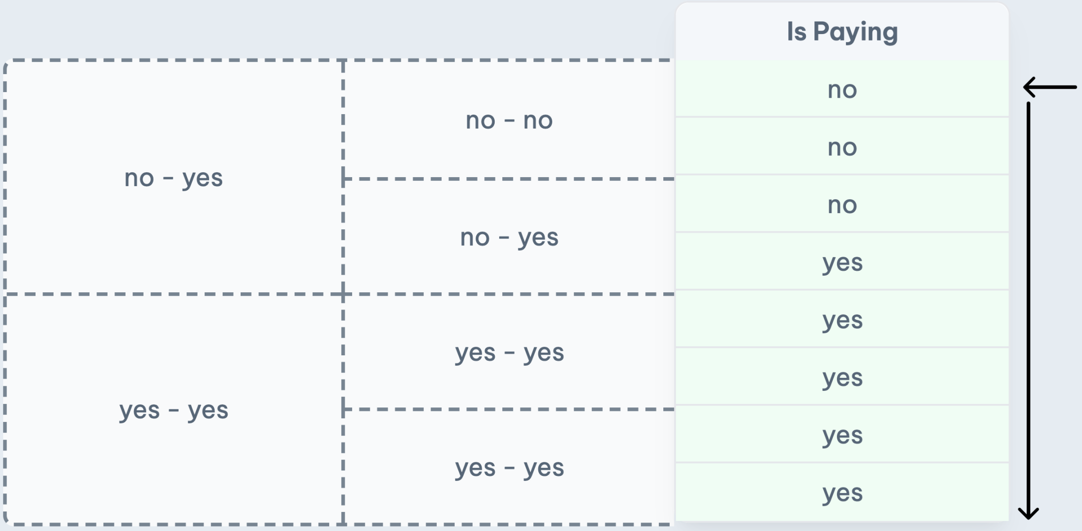
A Simple Group BY

```
SELECT is_paying, COUNT(*)
FROM users
GROUP BY is_paying
```

Similar values for the is_paying column should be stored consecutively one after another (Fig. A) so that the database can just loop over the index and count the values. The counter is incremented when the current loop's value is the same as the last one. But when a new value is discovered, the counting for the old value is finished and a new counter is started for the new grouping. This is very efficient and doesn't need any auxiliary temporary tables.

Figure A

Index on **(is_paying)**



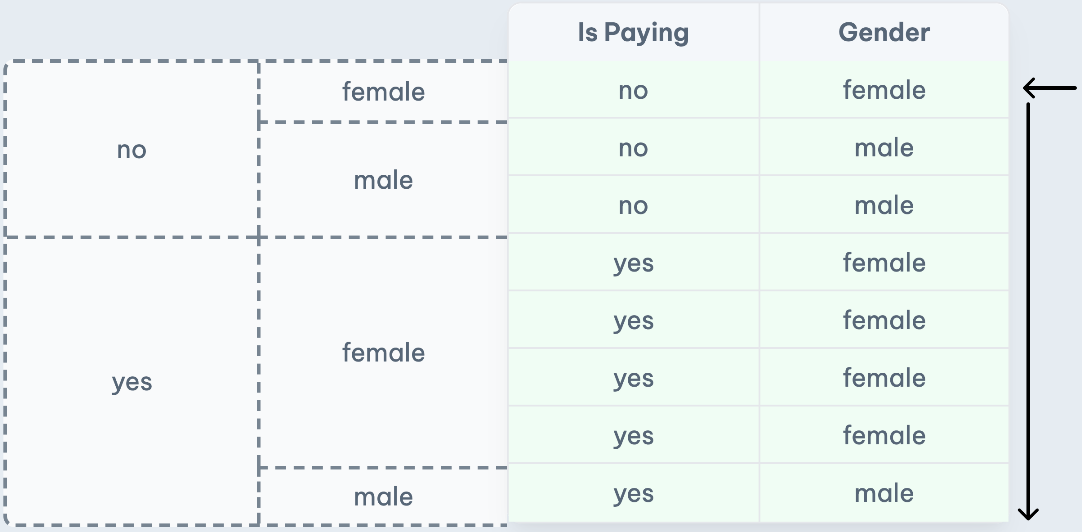
The query can be extended to group by multiple columns with different indexing requirements.

```
SELECT is_paying, gender, COUNT(*)
FROM users
GROUP BY is_paying, gender
```

When grouping by multiple columns, an index must always have the same columns in the same order. Therefore, the index must be created on the is_paying and gender columns used with the GROUP BY (Fig. B).

Figure B

Index on **(is_paying, gender)**



Is Paying	Gender
no	female
no	male
no	male
yes	female
yes	female
yes	female
yes	female
yes	male

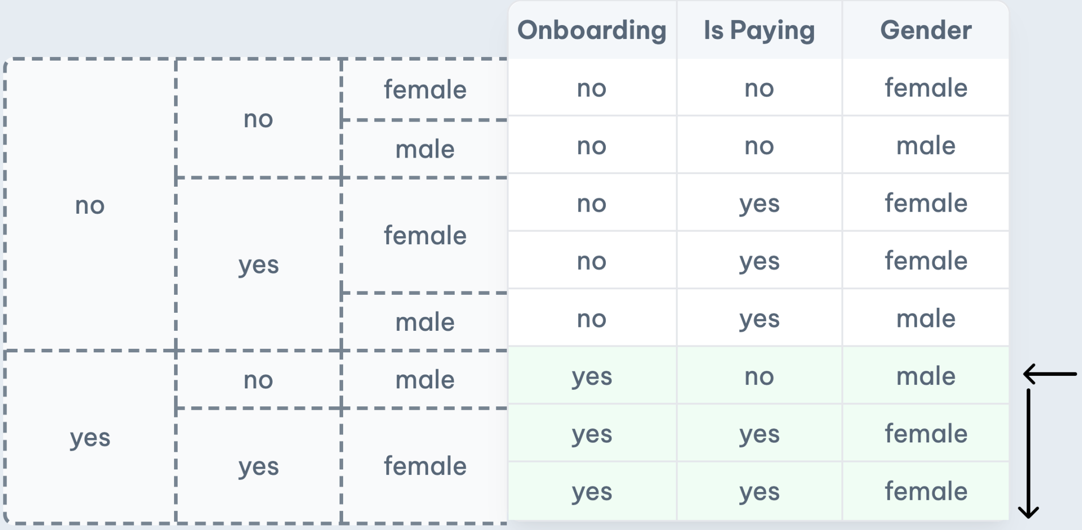
Adding WHERE Conditions

```
SELECT is_paying, gender, COUNT(*)
FROM users
WHERE onboarding = 'yes'
GROUP BY is_paying, gender
```

You must imagine how the database executes a query that filters the rows with a WHERE and applies a GROUP BY: The rows are filtered before the GROUP BY part is executed (SQL execution order). Therefore, the columns used in the WHERE part must always be added before the columns in the GROUP BY (Fig. C). By this specific ordering, the database can look up the first matching row and then scan forward to do the grouping.

Figure C

Index on **(onboarding, is_paying, gender)**



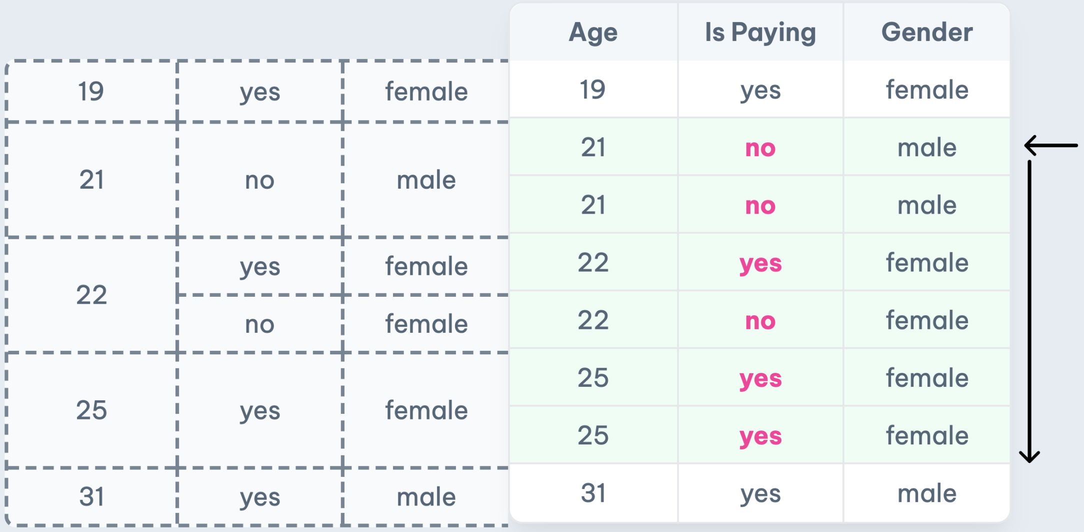
Onboarding	Is Paying	Gender
no	no	female
no	no	male
no	yes	female
no	yes	female
no	yes	male
yes	no	male
yes	yes	female
yes	yes	female

But what happens if the WHERE is extended to use a range condition and an index is created similar to before (Fig. D)?

```
SELECT is_paying, gender, COUNT(*)
FROM users
WHERE age BETWEEN 20 and 29
GROUP BY is_paying, gender
```

Figure D

Index on **(age, is_paying, gender)**



19	yes	female
21	no	male
22	yes	female
	no	female
25	yes	female
31	yes	male

Age	Is Paying	Gender
19	yes	female
21	**no**	male
21	**no**	male
22	**yes**	female
22	**no**	female
25	**yes**	female
25	**yes**	female
31	yes	male

The index is perfectly suitable for the WHERE condition but can't be used for the GROUP BY anymore: The grouping columns added to the index are intermingled because of the increasing age numbers. They are correctly sorted by the `is_paying` and `gender` columns for every `age` column but no longer form a consecutive block. The database can't apply the simple aggregation algorithm by just looping through the data anymore. Each new row in the loop may belong to a different group so a temporary mapping table is needed to store the intermediate results for each combination of `is_paying` and `gender`.

This query could again be optimized by transforming the age condition to a boolean value (*5.3. Transforming Range Conditions*).

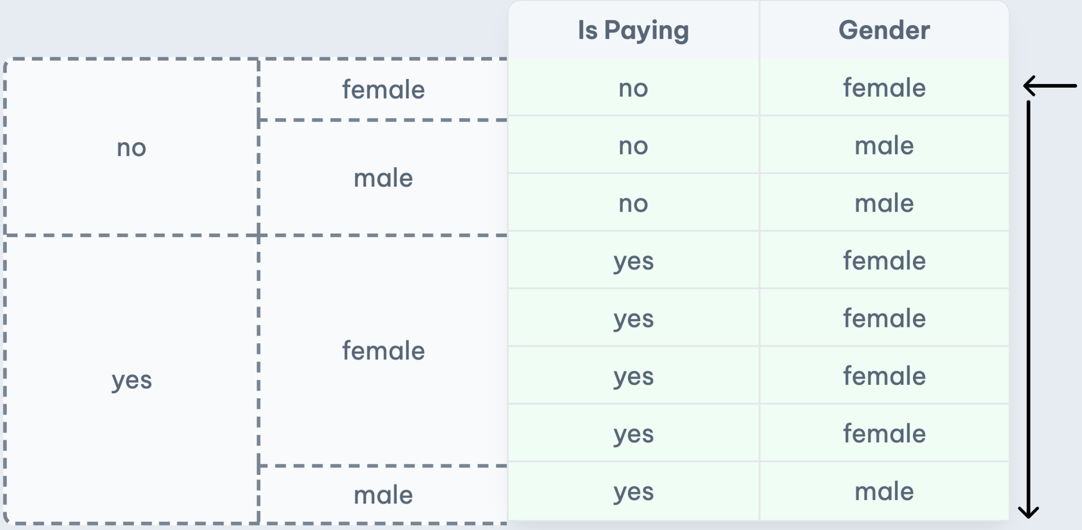
Using Aggregate Functions

```sql
SELECT is_paying, gender, avg(projects_cnt)
FROM users
GROUP BY is_paying, gender
```

Special care must be taken when using aggregation functions that calculate a set of values into a single value (e.g. `avg(projects_cnt)`). The former index (Fig. B) will be used, but it will not be efficient. As the `projects_cnt` column is missing from the index, the database has to load it for all rows matching the WHERE condition. This will be very slow if e.g. tens of thousands of rows remain after filtering.

Figure B

Index on **(is_paying, gender)**

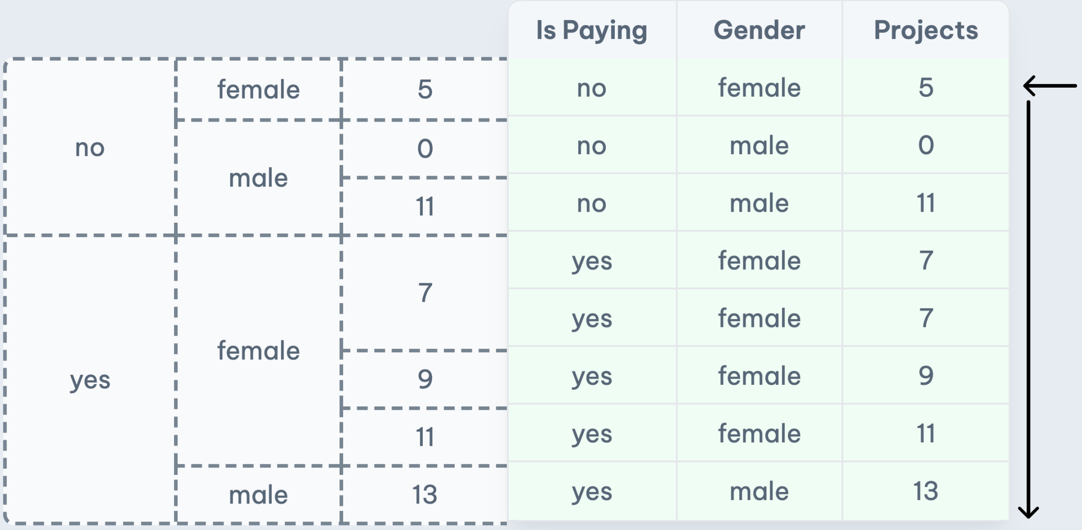


Is Paying	Gender
no	female
no	male
no	male
yes	female
yes	female
yes	female
yes	female
yes	male

The used columns in the SELECT part should be added to the index last (Fig. E) so that filtering and grouping work efficiently and the columns to be aggregated are available without loading the rows.

Figure E

Index on **(is_paying, gender, projects_cnt)**



Is Paying	Gender	Projects
no	female	5
no	male	0
no	male	11
yes	female	7
yes	female	7
yes	female	9
yes	female	11
yes	male	13

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!

```sql
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.

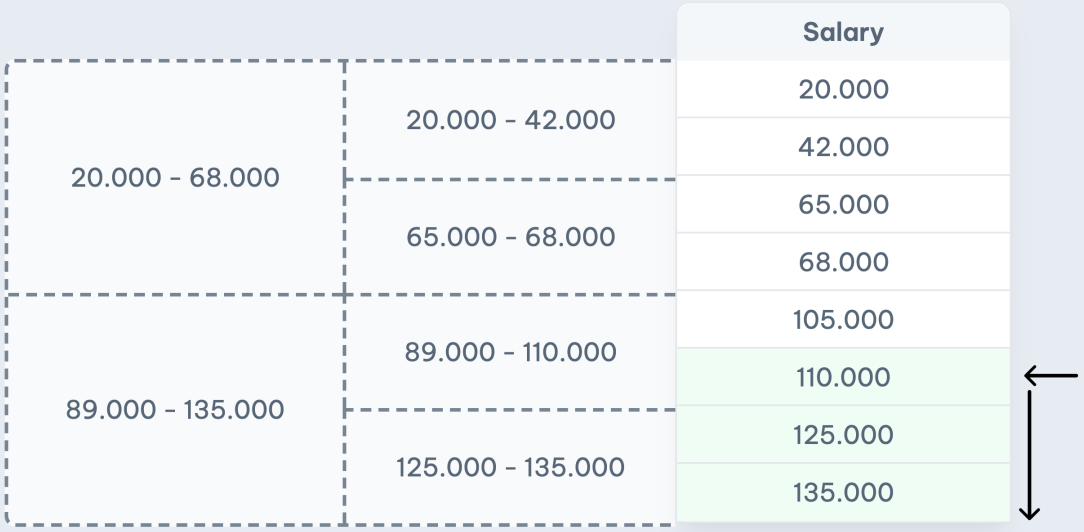
```sql
SELECT *
FROM employee
WHERE salary > 100000;

-- 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

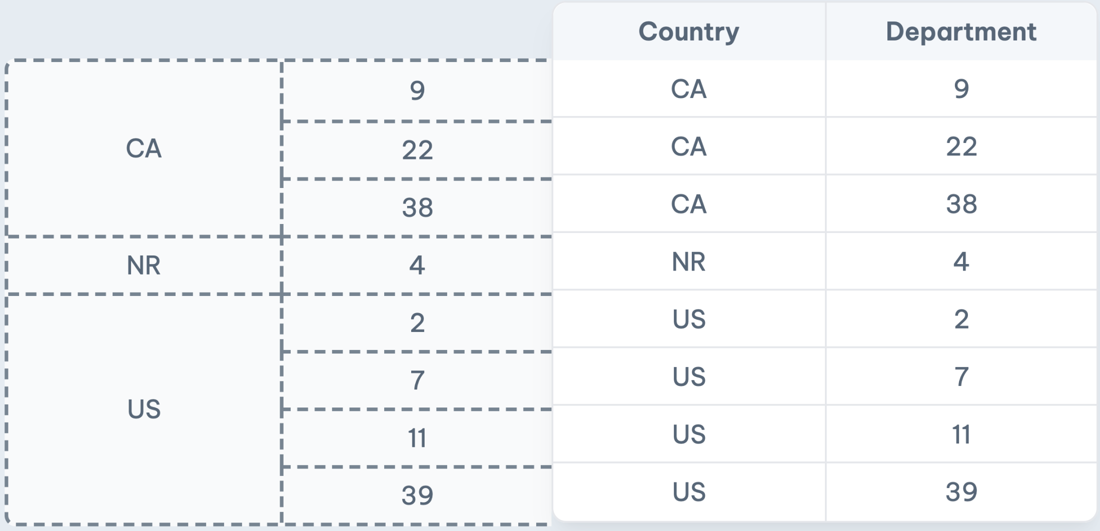
Index on **(salary)**



Salary
20.000
42.000
65.000
68.000
105.000
110.000
125.000
135.000

Figure B

Index on **(country, department_id)**



Country	Department
CA	9
CA	22
CA	38
NR	4
US	2
US	7
US	11
US	39

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.

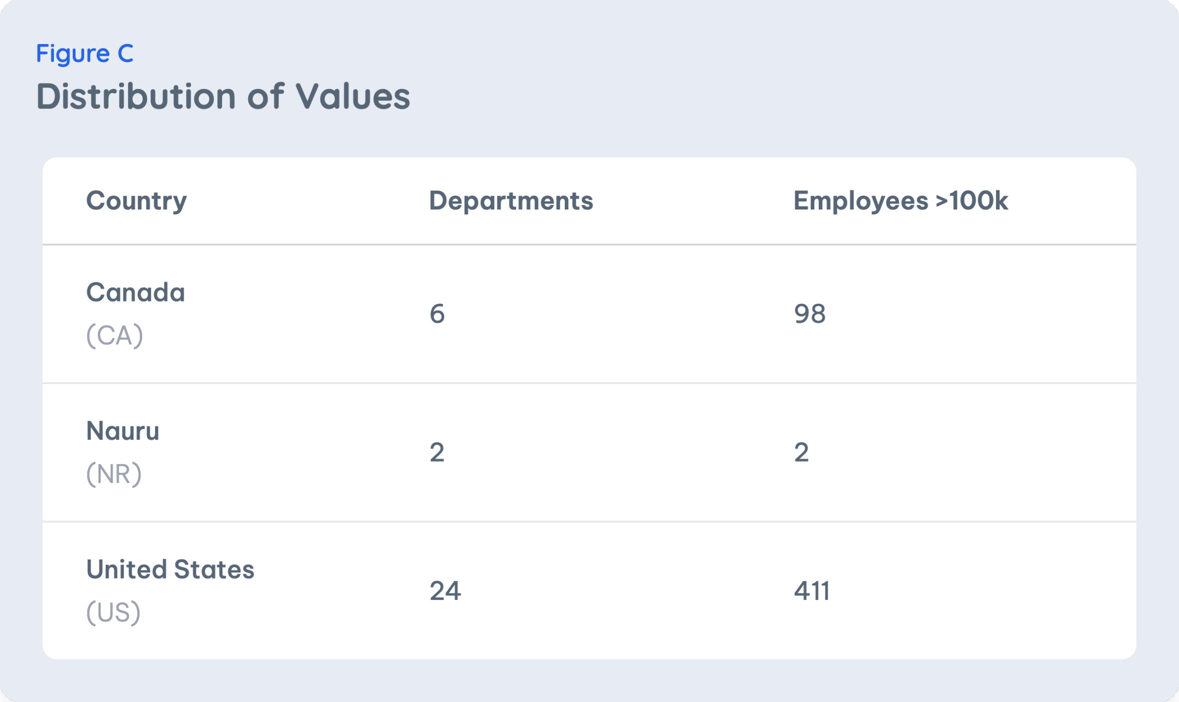


Figure C

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?

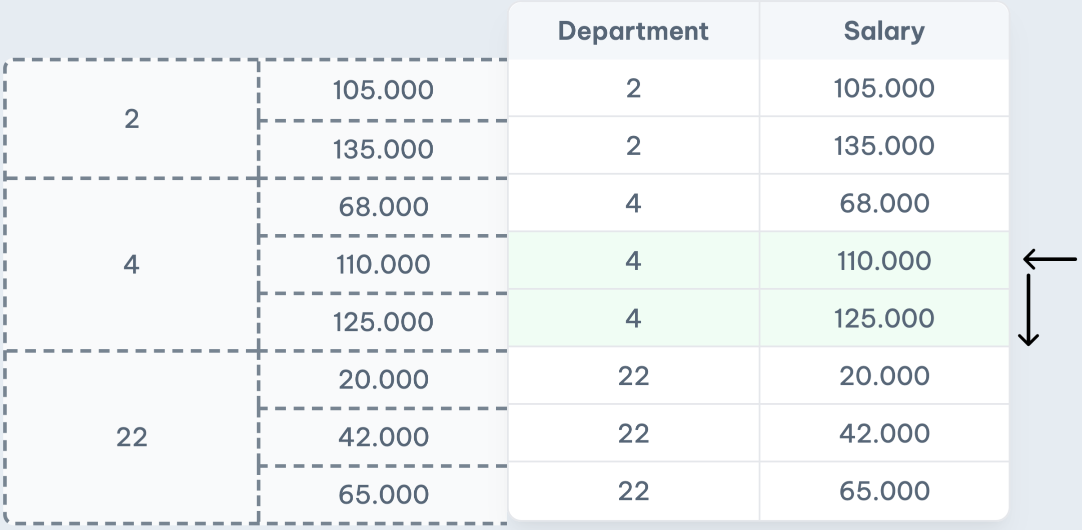
```sql
SELECT *
FROM department
WHERE country = 'NR';

-- 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)**



Department	Salary
2	105.000
2	135.000
4	68.000
4	110.000
4	125.000
22	20.000
22	42.000
22	65.000

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.

3.7 Subqueries

Subqueries are still believed to be slow, although they are just missing matching indexes. Creating good ones for subqueries is not particularly complicated: You always optimize them one by one independently, whether they are used e.g. in WHERE to do more fancy filtering or in SELECT to add new columns based on data from other tables.

It is essential to understand the difference between independent and dependent subqueries as both have different requirements for a good index.

Independent Subqueries

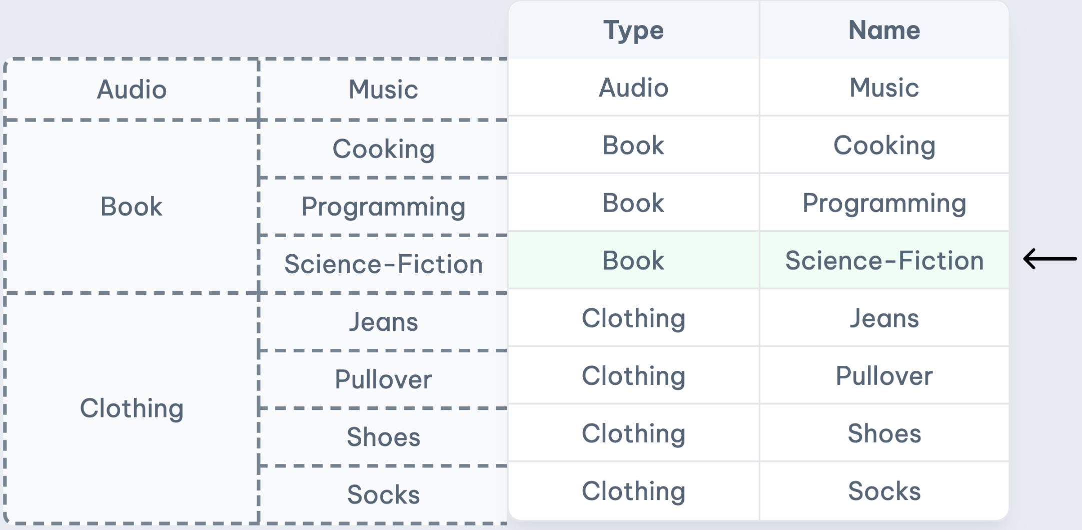
```sql
SELECT *
FROM products
WHERE remaining > 500 AND category_id = (
  SELECT category_id
  FROM categories
  WHERE type = 'book' AND name = 'Science fiction'
)
```

The query searches for products within a specific category and huge remaining stock to free up storage space by selling them with a considerable discount. This is an independent subquery because no tables from the query surrounding the subquery are used within the subquery. It is executed independently only once and the condition of the surrounding query is rewritten to use the resulting `category_id` from the subquery.

Indexes for independent subqueries are created by ignoring that they are part of a much more complex query. With this approach, the subquery index is created using the `type` and `name` columns (Fig. A).

Figure A

Index on **(type, name)**

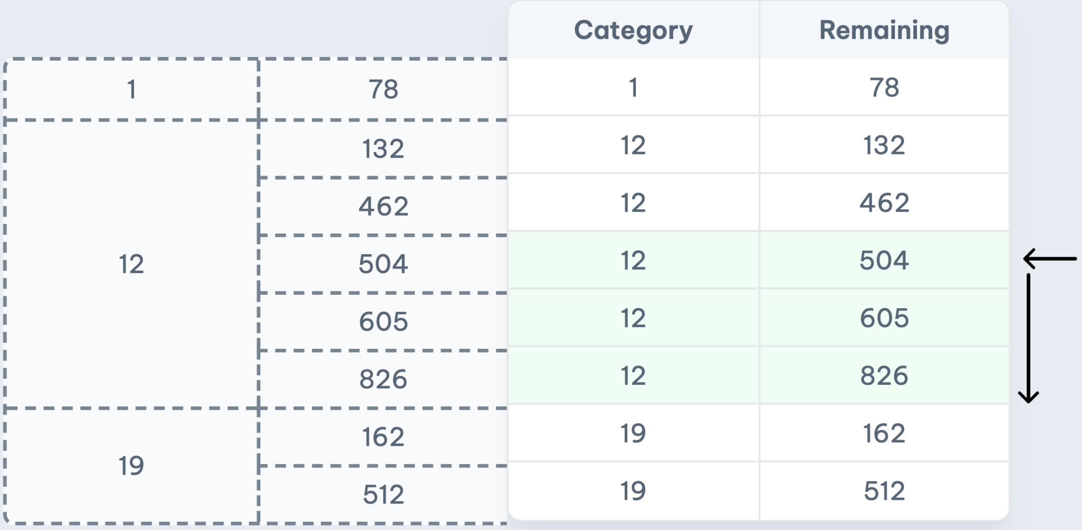


Type	Name
Audio	Music
Book	Cooking
Book	Programming
Book	Science-Fiction
Clothing	Jeans
Clothing	Pullover
Clothing	Shoes
Clothing	Socks

The subquery used for the example yields precisely one value as expected by the outer query. So when that value has been computed, it replaces the subquery in the SQL statement. The index is now built for a simple query with only two equality conditions. With the range condition for `remaining` in mind, the final index uses the `category_id` and `remaining` columns (Fig. B).

Figure B

Index on **(category_id, remaining)**



Category	Remaining
1	78
12	132
12	462
12	504
12	605
12	826
19	162
19	512

Dependent Subqueries

```
SELECT *
FROM products
WHERE remaining = 0 AND EXISTS (
  SELECT *
  FROM sales
  WHERE created_at >= '2023-01-01' AND product_id = products.product_id
)
```

The query searches for products with empty stock and recent sales to reorder them again. This is a dependent subquery because it references a table from the surrounding query - it depends on the `product_id` column from the `products` table.

The indexes for dependent queries are created like joins: The surrounding query is executed first and only needs an index on the `remaining` column (Fig. C). The subquery is executed repeatedly for each matching row of `products` with the `products.product_id` column being a different value each time. The suberquery's index (Fig. D) has to use the `product_id` and `created_at` columns.

Figure C

Index on **(remaining)**

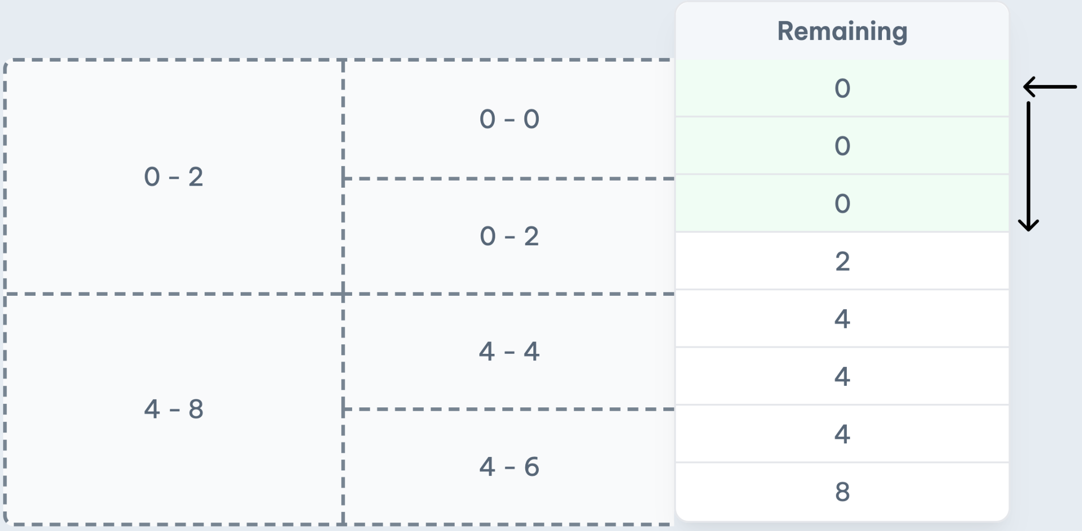
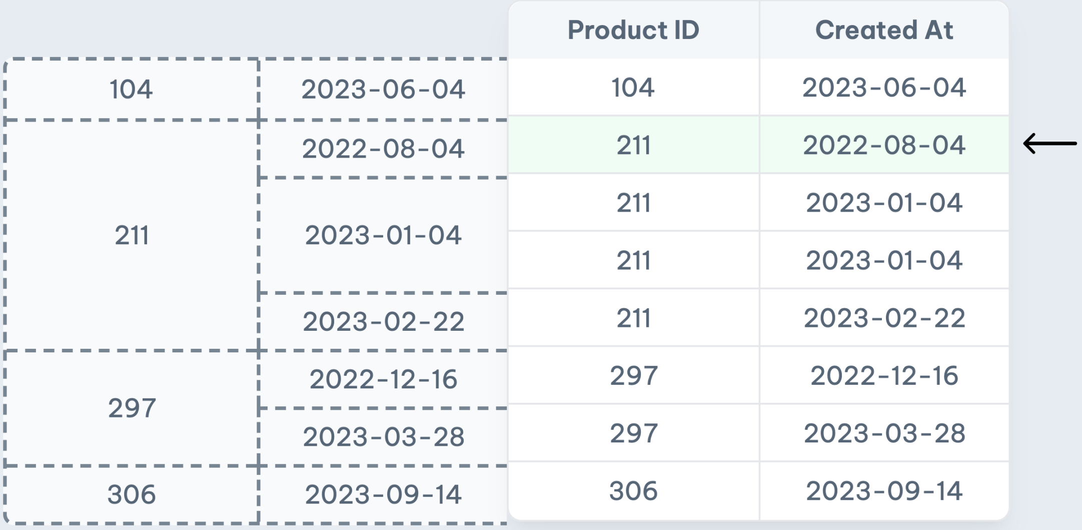


Figure D

Index on **(product_id, created_at)**



Product ID	Created At
104	2023-06-04
211	2022-08-04
211	2023-01-04
211	2023-01-04
211	2023-02-22
297	2022-12-16
297	2023-03-28
306	2023-09-14

The index access for the subquery is stopped after finding the first result because the EXISTS condition of the surrounding query is already satisfied with one matching row. Scanning all matching index entries would be a wasted effort. You can imagine that a subquery wrapped in `EXISTS()` always has a `LIMIT 1` applied automatically. Nevertheless, the index must be built according to range condition principles (*Index Access Principle 4: Scan On Range Conditions*) for the best efficiency.

3.8 Data Manipulation (UPDATE and DELETE)

Many times neither UPDATE nor DELETE queries are considered for optimization. Those queries are allowed to be much slower because they have to modify rows rather than just read them. The slower execution time is valid but is only part of the truth.

Both have to find matching rows before they can modify or delete them. You can imagine them as a SELECT query that does not return the rows at the end but modifies them. And the part of finding the rows can be optimized by every principle and trick the same as SELECT queries. You can rewrite them into SELECT queries and optimize them.

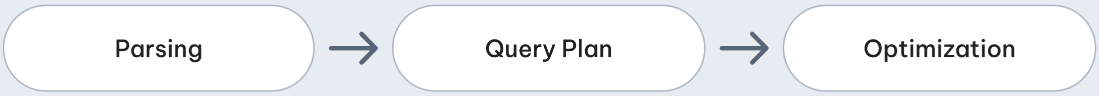
4. Why Isn’t the Database Using My Index?

It is always a frustrating experience when a recently created index is not used. You only want to make one query faster. But for some reason, you can't get the result you want to have.

You should know what to look out for if you discover that an index is not used. There are many different reasons, all of which have unique causes.

Figure A

Query Execution Steps



Understanding how a query is executed is essential to know why the index you expected to be used is ignored. Each query follows the process illustrated in Fig. A:

1. The query is first parsed, i.e. split into its components, so that the intention behind the query can be identified.
2. A simple plan is created for the execution of this query. At this stage, all operations are still full-table scans because they can always be executed - no matching indexes etc. are required. The idea is to always start with a possible way of execution.
3. After creating the initial query plan, it is optimized. The indexes are checked to determine whether they would make the query faster and many other optimizations are evaluated. It is important to remember that the former query plan is kept if no better optimization has been found.

So if a query is executed differently than expected, the optimizer did make a different decision than you expected. The possible reasons for this are explained in the following chapters.

4.1 The Index Can't Be Used

The most common error is an index that does not match the query: The query optimizer will never consider an index if it does not apply to the query.

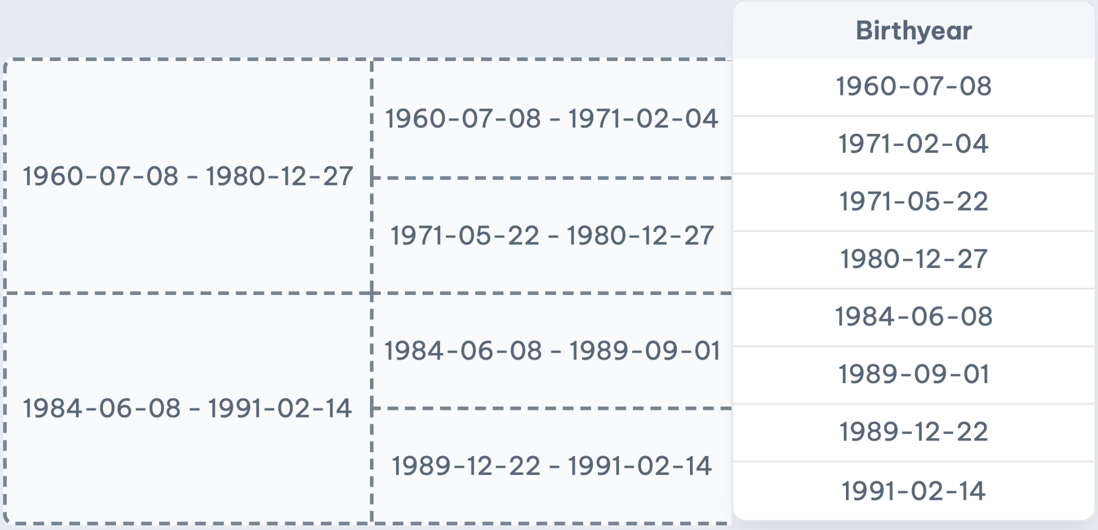
Column Transformations

A widespread problem is transformations to a column. Any change to a column by an operation causes an index to no longer be used. Common examples are conditions like `YEAR(birthdate) = 1970`, `col + 5 < 20` or `CONCAT(firstname, ' ', lastname) = 'Tobias Petry'`.

Let's take a closer look at the example using the `year(birthdate) = 1980` condition: Looking at the index (Fig. A) of such a column, it is obvious that it refers to a whole date and not just the year part. However, you can argue that the database should recognize that you are only interested in the year and the query should be automatically rewritten to `birthdate BETWEEN 1980-01-01 AND 1980-12-31` so that it can use the index. After all, the query optimizer's mission is to optimize a query and utilizing an index is its most important task.

Figure A

Index on **(birthyear)**



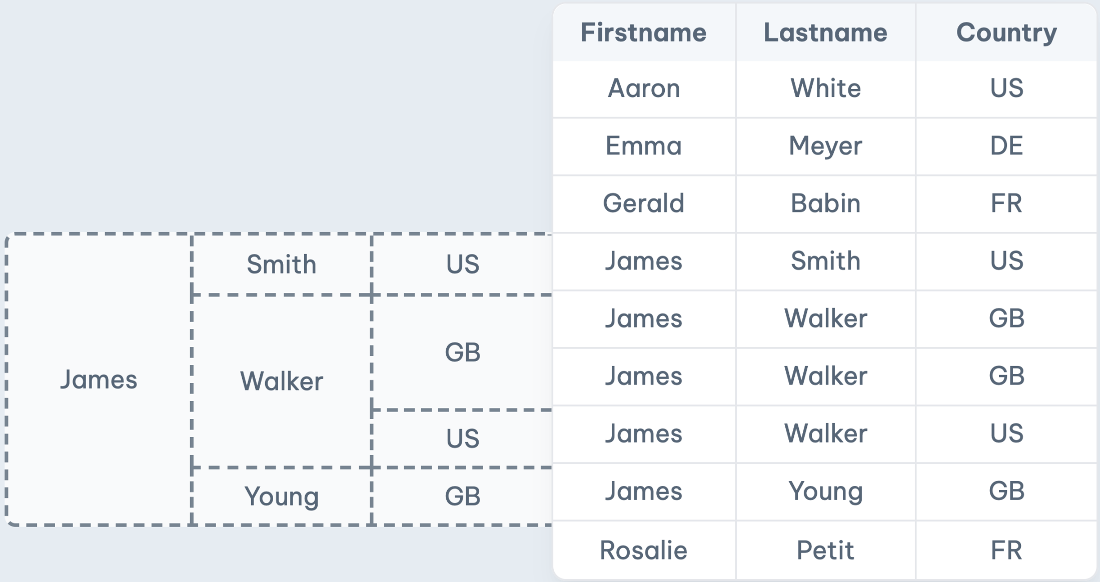
This opinion is indeed correct. However, it would massively increase the complexity of the database implementation since special logic must be prepared for every possible transformation. This is complicated to even impossible if several transformations are applied to a column one after the other. The cases in which such automatic query rewriting could work and not work would therefore be hard to understand. For this reason, databases do not implement this kind of optimization to keep the basic principle always the same: An index will never be used if a transformation is applied to the indexed column. But you can create an index on the column's transformed value: `CREATE INDEX contacts_birthdate on contacts ((YEAR(birthdate)))` (chapter "*5.1 Indexes on Functions*").

Incompatible Column Ordering

As described in chapter "*2.3. Index Access Principle 3 - From Left to Right*", the column order in a multi-column index is essential: A condition on the column `country` can only be used with the index of Fig. B if the query also uses the column `firstname`. However, it would be better to use the column `lastname` as well.

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

Operations Not Supported by the Index

Another error source is assuming that an index can be used for an operation it is not designed for: An index on `firstname` can't be used for a condition such as `firstname like '%Tobias%'`. The details of this problem are described in the chapter "*5.4 Leading Wildcard Search*. Beyond that, there are no noteworthy pitfalls in this category.

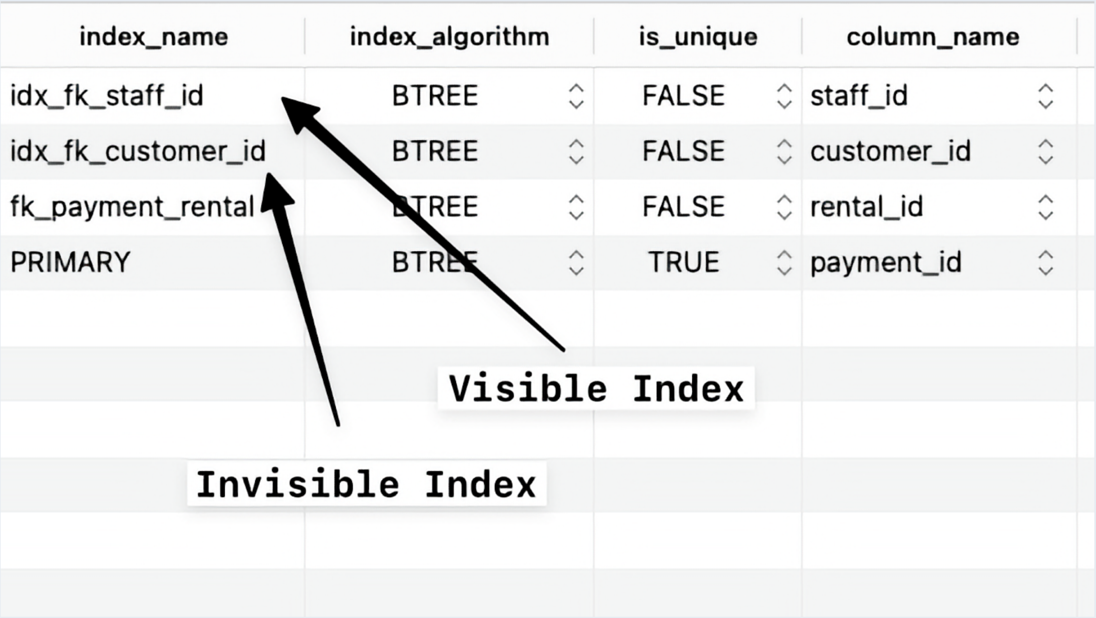
However, PostgreSQL has many more operators apart from the normal comparison functions. For example, you can check whether keys exist in a JSON object (`attributes ? 'key'`), whether two date ranges overlap (`checkin_checkout_time && tsrange(start, end)`) and many more. All these new querying possibilities can use an index but require either a particular type of index or specific index properties to be set. The exact use should be taken from the related chapter of the PostgreSQL documentation since the list of new operators is too big to be handled here.

The Index Is Invisible

As you can never be entirely sure if an index is still being used by some query, they are often never deleted. Even one slow query could seriously impact the application and re-creating the deleted index will take a long time for large tables. So with MySQL (but not PostgreSQL) you can make indexes invisible before deleting them - and reactivate them in a few seconds if needed.

Figure A

Invisible Indexes Are Not Identifiable



index_name	index_algorithm	is_unique	column_name
idx_fk_staff_id	BTREE	FALSE	staff_id
idx_fk_customer_id	BTREE	FALSE	customer_id
fk_payment_rental	BTREE	FALSE	rental_id
PRIMARY	BTREE	TRUE	payment_id

However, these invisible indexes are not displayed correctly by all desktop tools (e.g. TablePlus): Some continue to show them without indicating that they are hidden. When you want to use this feature, check the behavior of your tools for this specific edge case or delete indexes shortly after making them invisible.

4.2 No Index Will Be the Fastest

Sometimes you've made none of the former explained mistakes with your indexes and queries: The query perfectly matches an index's capabilities but is still not used. These issues seem to be strange but are easy to explain as they fit into a few very clearly defined categories. However, you first need to learn a bit more background knowledge.

How an Index Is Selected

Every database stores statistics about the columns of a table with their possible values or value ranges, as well as the frequency of occurrence.

Figure A

Value Distribution with Percentages

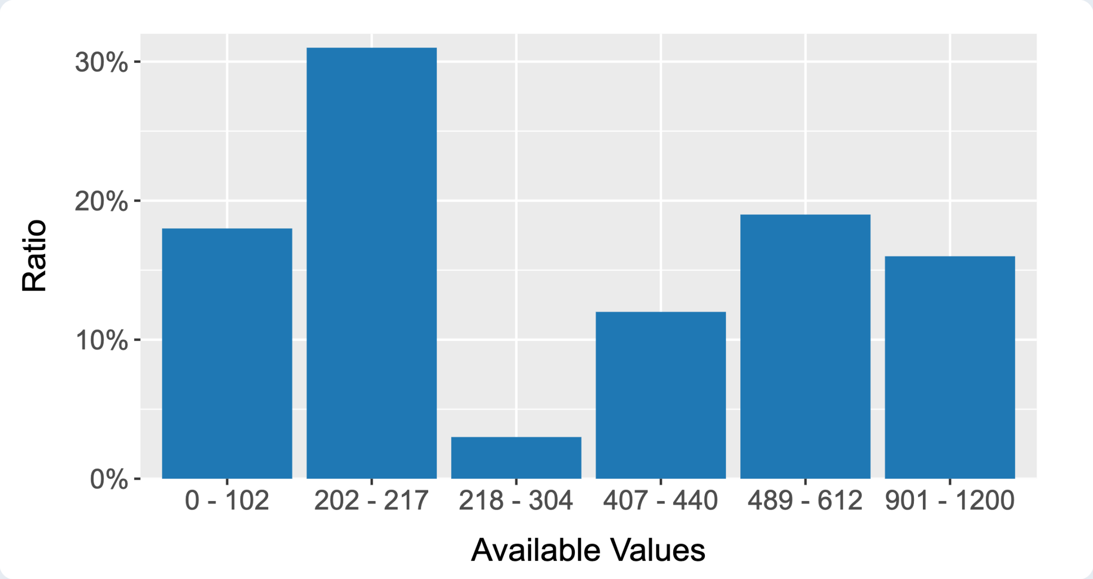
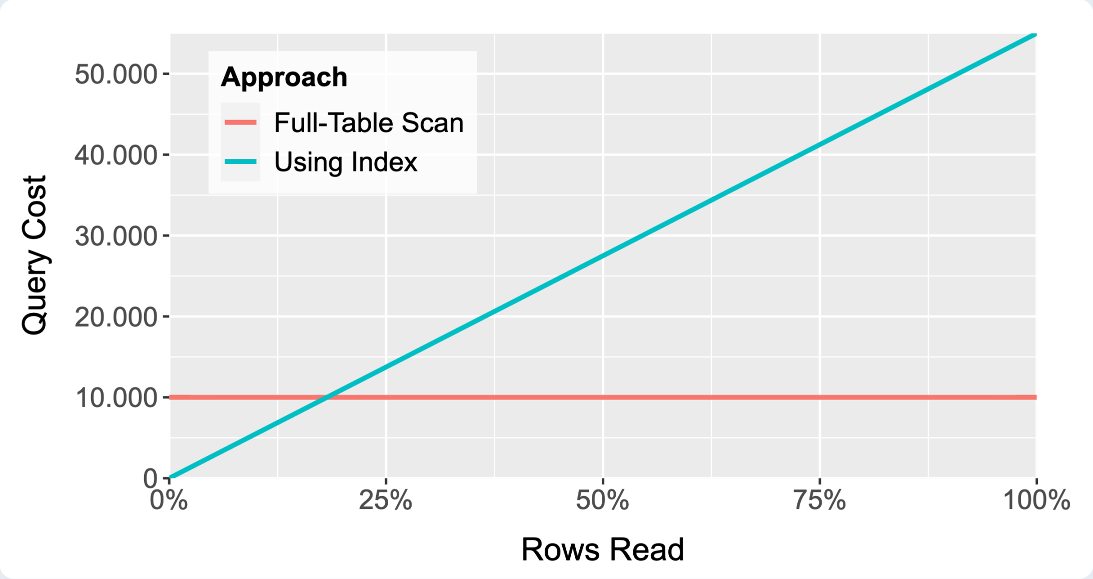


Figure A visualizes the data distribution statistics of an example column. These statistics are calculated by inspecting a set of random rows to have a reasonable sample. Of course, not all rows can be considered as this would take a long time for tables with millions or billions of rows. That not all rows are included in the statistics is not a problem. The sample size is sufficient to extrapolate the values to the entire table and undiscovered values can be calculated to exist at most until a certain percentage. The logic is very complex but works exceptionally well.

These column statistics are used to make predictions about performance. They are the source for estimating how many table rows will match the query's conditions and which approach will be the fastest.

Figure B

Query Costs for Different Approaches



The performance rating for different approaches is calculated by using a cost model. Each operation that must be performed is assigned an abstract cost value that closely describes its impact. They are fine-tuned to be comparable but don't resemble an exact correlation to the time spent or any other measurable metric. For example, Fig. B shows a simplified query cost model on a table with 10,000 rows. The predicted costs for a table scan or using an index are significantly different. A different approach must be chosen depending on how many rows will likely match the conditions. Therefore, the query optimizer uses the cost model to rate many possible query optimizations and selects the most efficient one.

The costs shown here are exemplary and are in reality much more complicated. In fact, the developers of databases spend a great part of their time constantly improving the cost model and optimizations. As the more advanced the cost model is, the better even the most complicated queries are optimized.

Loading Many Rows

When an index is used, the matching index entries identify the rows to load. They are then loaded one after another from the large data file of the table (chapter “*1.2 The Interaction of Indexes and Tables*”). But this is complicated because these rows are all stored in the file at different positions. So for each row, it is necessary to jump to a different offset in the file. This is called random i/o because the data must be loaded from many seemingly random locations.

Old hard drives with spinning disks could only perform a few such jump operations per second. However, modern SSDs are much faster at this but still orders of magnitude slower than simply loading data sequentially (sequential i/o).

As already seen in the above cost model, there is a specific threshold when iterating the entire table's file sequentially is faster than loading only the needed rows with random i/o: A query that loads at least 10%-30% of the rows in a table will usually be faster with a full-table scan. This limit is not exact and can shift depending on the table size (e.g. millions of rows) or the columns (e.g. the amount or size of columns). Nevertheless, it is a reasonable estimate you should keep in mind.

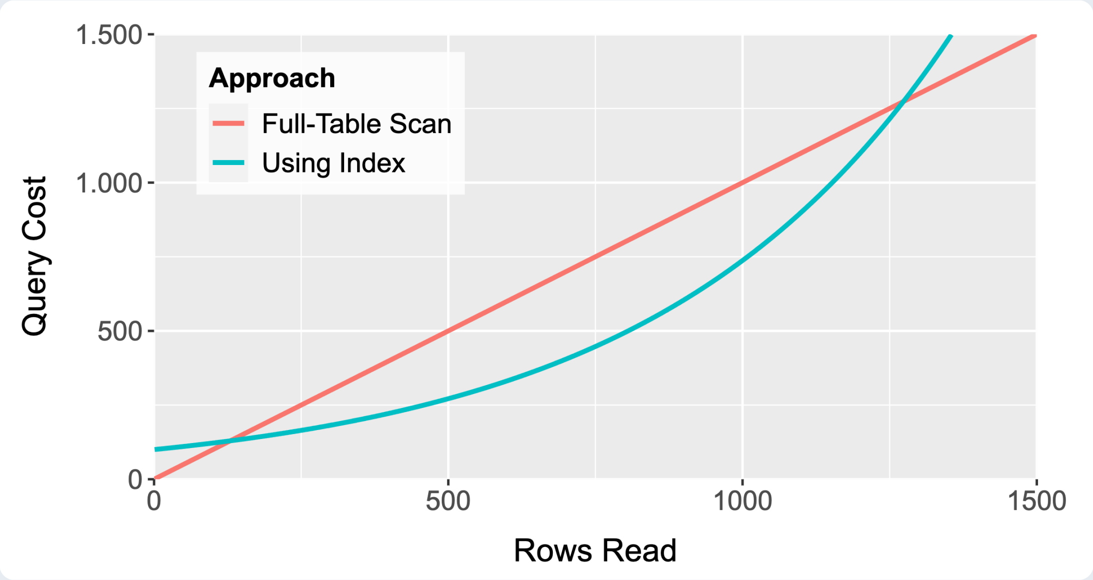
> **Tuning Advice**
>
> In PostgreSQL, you can tune the configuration to tell the query optimizer you are using a fast disk: You should set `random_page_cost` to `1.1` if your database fits entirely into memory or you use an SSD.

Small Tables

The previous cost model was simplified to explain the general concept but didn't include all the fine details of accurate cost models. When tables are small (e.g. 100-200 rows) just loading all rows sequentially can be faster than using an index and loading only the required ones with random i/o (Fig. C).

Figure C

Query Cost for Loading a Small Number of Rows

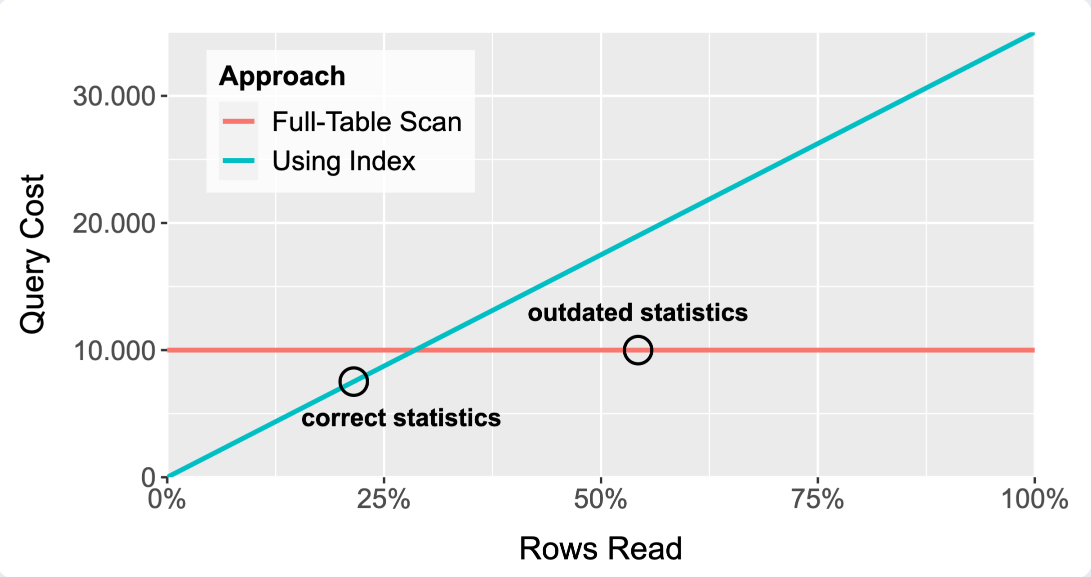


Outdated Statistics

The column statistics are generated once but are not dynamically adjusted. They will no longer match the actual distribution after e.g. mass updates or many new rows are inserted or deleted. However, every database has a heuristic to decide when the statistics are outdated and must be recalculated. This happens completely in the background without you having to do anything.

Figure D

Query Cost with Wrong Row Estimation



But you have a problem if the heuristic has not yet been triggered and the statistics differ significantly from the actual distribution. The number of matching rows of a database query will now be over- or underestimated and the cost model will be negatively influenced. This can be seen in the example of Fig. D: With outdated statistics, it is estimated that slightly more than 50% of the rows match a query and a full table scan is the best approach. In reality, fewer than 25% match the query and using an index would be better. Significantly skewed statistics can lead to bad optimizations.

> **Expert Advice**
>
> I always recommend to trigger a recalculating of the statistics after bulk changes to tables using ANALYZE TABLE. Another frequently used solution is using a cron to consistently recalculate the statistics e.g. hourly.

4.3 Another index is faster

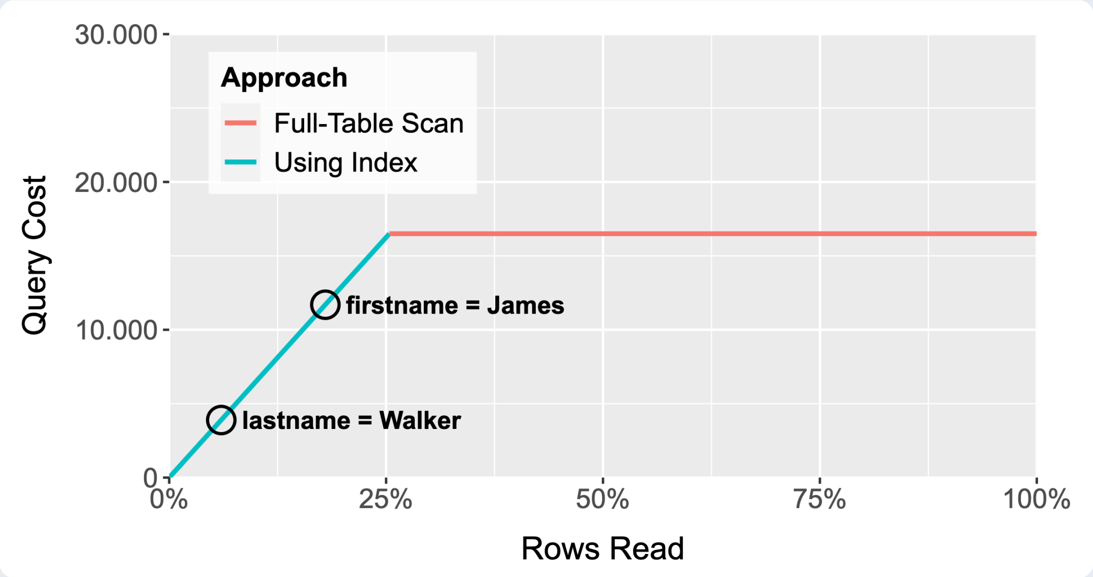
The database can also select a different index than you expected because of the cost model. There are many different manifestations of this behavior. Therefore, only the most frequent ones will be discussed here to give an impression.

Multiple Conditions

A multi-column index is best for a query with the condition `firstname = 'James' and lastname = 'Walker'`. However, frequently applications only have single-column indexes. So the database must decide must select the one with the lowest for the query. A different index may therefore be favorable depending on the varying values in a condition (Fig. A).

Figure A

Costs for Specific Indexes



Joins

Optimizing joins is much more complex. As explained in chapter "*3.6 Joins*", the query optimizer can change the join order to do fewer operations. For this optimization, the costs of loading rows and the costs of loop cycles must be optimized. It is possible that a specific join order is always more efficient (Fig. A) or must be decided depending on the conditions (Fig. B).

Figure B

Specific Join Order Is Always Faster

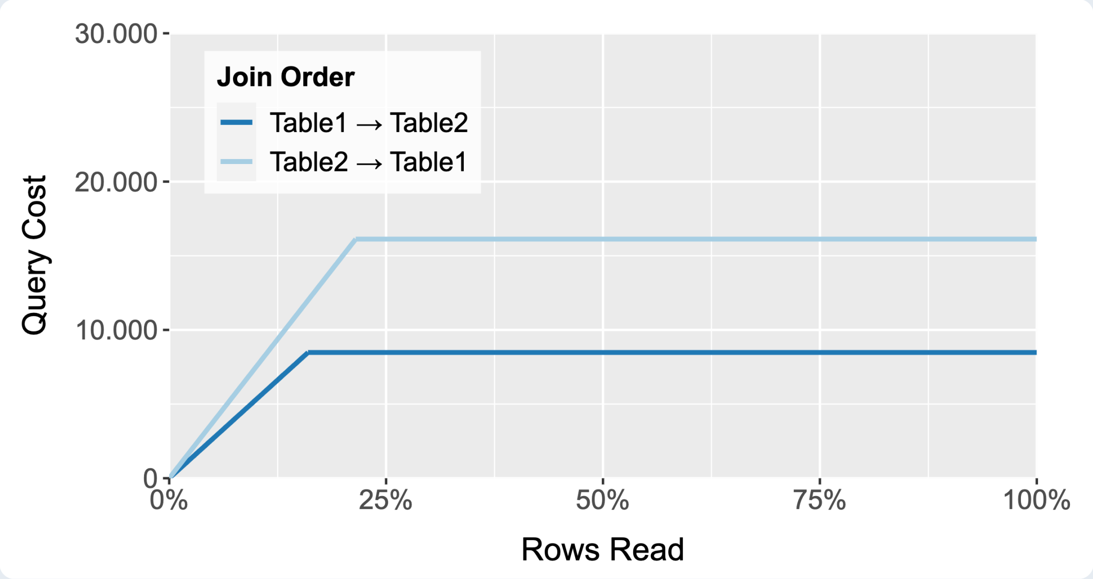
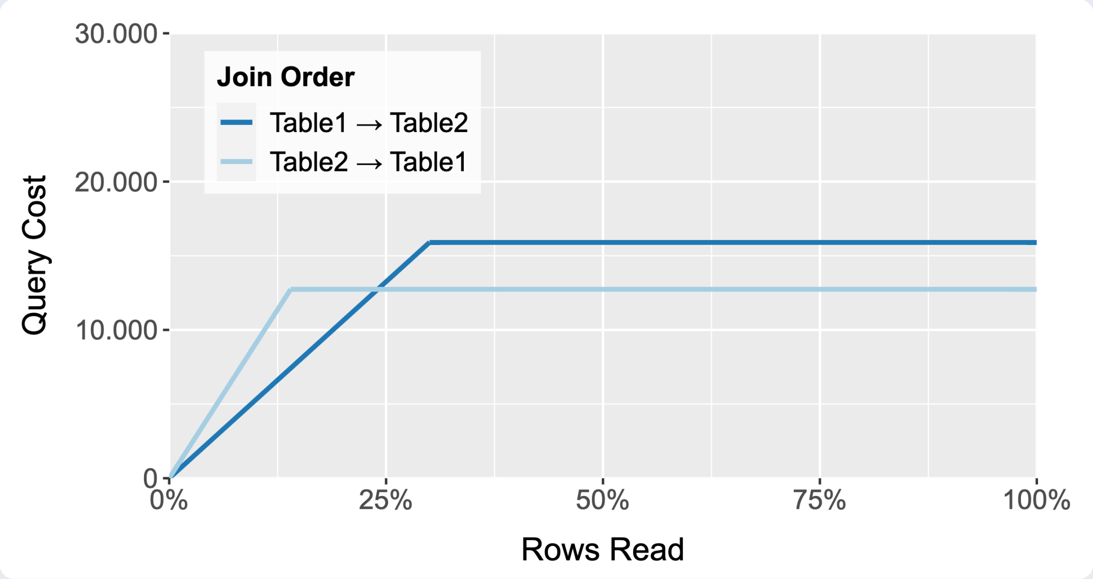


Figure C

Join Order Depends on Specific Query Values



Additionally, the number of joins is also important. A join with four tables yields 24 orders to execute it - multiplied by the possible number of indexes. With many tables, the optimization becomes more and more difficult as not all permutations can be calculated. So you should only join a couple tables and not a dozen ones to get excellent and predictable performance.

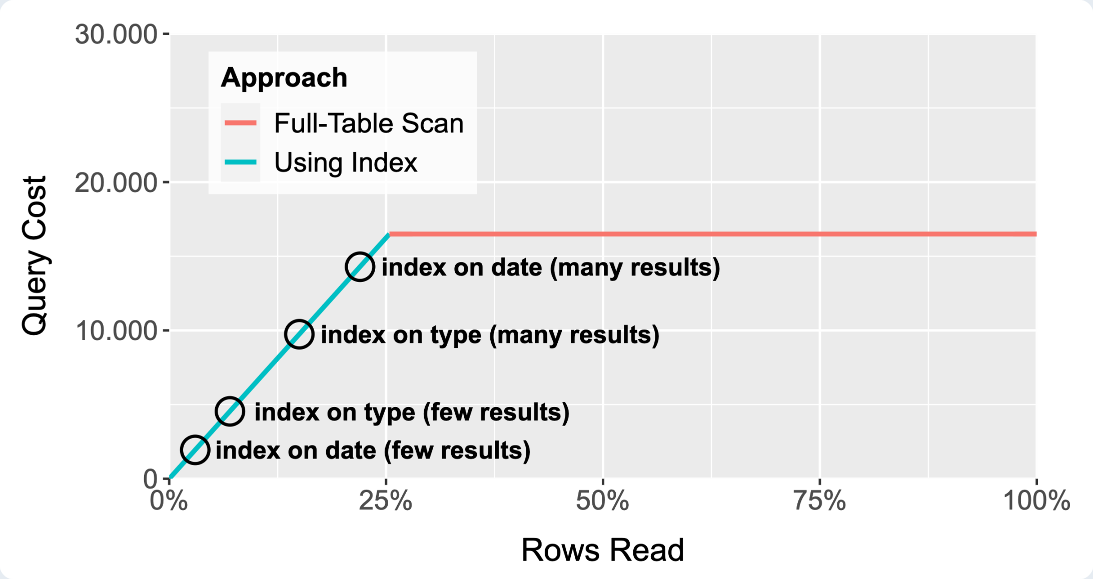
Ordering

Most tables don't have perfect indexes that simultaneously speed up filtering and sorting. For a simplified example, there may be an index on `type` and one on `date` for the query `WHERE type = 'open' ORDER BY date DESC LIMIT 10`. The database will have two different ways to execute this (Fig. D):

1. The database will use the index on `type` and then do an additional sorting step. This makes sense, as the number of open issues is significantly lower than the closed ones and will probably fit into memory. The high filtering ratio of the condition led the database to use this approach.
2. However, the database could also use the `date` index to read the rows in sorted order. Loading many rows by the filtering index and sorting them could be a slower approach than loading in sorted order and only keeping the matching ones. The critical point is how many rows must be loaded for each approach and how much sorting would cost.

Figure D

Query Cost for Different Indexes



5. Pitfalls and Tips

Understanding the fundamentals of how indexes are used and utilized by every SQL operation is essential to create good indexes. But there are still a few things to be aware of beyond the indexing principles and knowing when indexes are not used.

You will encounter some pitfalls with real applications that you may do occasionally, even if you know all the rules. They are problems that are easy to overlook and should be focused on.

Apart from possible mistakes, there are still some tricks for indexing. They can make your life much easier and simplify certain queries. Knowing them is beyond the basics but is absolutely essential.

5.1 Indexes on Functions

As chapter "*4.1 The Index Can't Be Used*" explains, an index on `birthday` cannot be used for a condition like `WHERE year(birthday) = 1988` because the column was transformed. This exemplary condition can be rewritten to an equivalent one using an index (`WHERE birthday BETWEEN '1988-01-01 AND 1988-12-31`), but it is impossible to do for all transformations. The transformation `WHERE month(birthday) = 5` cannot be rewritten to involve an index.

For this purpose, most databases (MySQL, PostgreSQL, SQLite, Oracle, etc.) have so-called functional indexes: An index is created on a transformation (e.g. by a function call) instead of the column directly. They are created identically to regular indexes by using a transformation wrapped in parentheses (to indicate them as functional indexes) instead of a regular column. The index will be used automatically if the exact same transformation is used in a query.

```
CREATE INDEX contacts_birthmonth ON contacts ((month(birthday)));

SELECT * FROM contacts WHERE month(birthday) = 5;
```

Virtual Columns

Not every database supports functional indexes. Some databases, such as MariaDB or Microsoft SQL Server require a more manual approach: You have to create a new virtual column for the transformation instead of being able to create an index on it. Take a look at the following example of how it is done with e.g. MariaDB:

```
CREATE TABLE contacts (
    id bigint PRIMARY KEY AUTO_INCREMENT,
    birthday datetime NOT NULL,
    birthday_month datetime AS (month(birthday)) VIRTUAL NOT NULL,
    INDEX contacts_birthmonth (birthday_month)
);
```

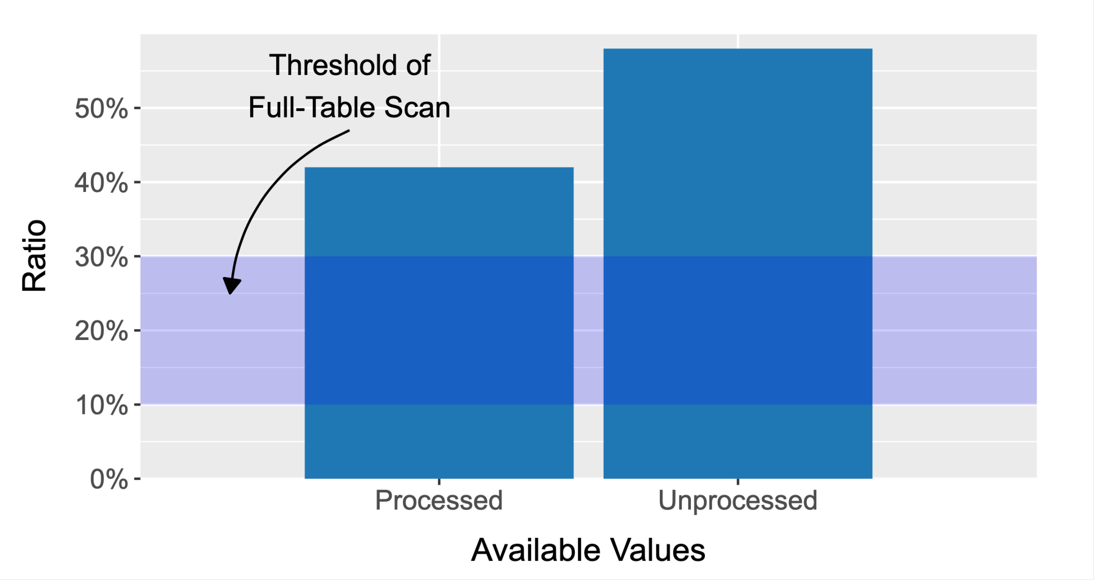
The usage of these generated columns is also different for every database. With e.g. Microsoft SQL Server, you can use the transformation in a query (`WHERE month(birthday) = 5`) and it will automatically use an index created on a matching virtual column. But this does not work for e.g. MariaDB: You have to use the newly generated column (`WHERE birthday_month = 5`) instead of the transformation in every query.

5.2 Boolean Flags

A common problem is indexing columns with boolean values. No index is used if the distribution of values is similar to 50:50 or a mere 80:20. This decision is again due to the cost model (“*4.2 No Index Will Be the Fastest*”): The number of matching rows for each value storing an unprocessed or processed state exceeds the threshold range for a full-table scan being faster than using indexes (Fig. A). After a time, fewer rows with unprocessed status remain and an index will be used. But until falling below the threshold every query is a slow full-table scan.

Figure A

Uniform Distribution With Boolean Values

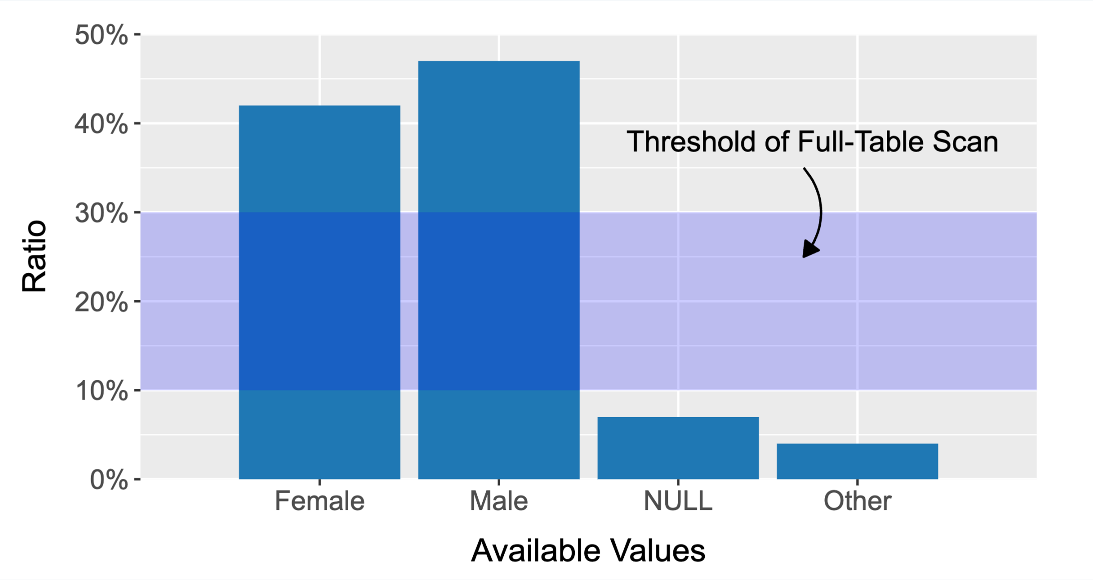


But even no index will be used when loading just a few rows with `WHERE status = 'unprocessed' LIMIT 5`. Let's assume a distribution of 80:20 with only 20% of the rows matching the condition. An index could be used here, but statistically, only 100 rows must be loaded on a full-table scan on average to find five matching ones. Using an index and loading those rows somewhere from the table takes longer than just loading 20 times more rows sequentially. Of course, the values in the table do not have to be equally distributed. The 20% matching values could be at the very end but the database does not know this and therefore can't consider this when calculating the costs.

The behavior is also identical for columns that do not store boolean values but have an uneven distribution. For example, a gender column will store many identifiers, including a NULL value for missing information. This is illustrated by Fig. C, which combines several gender values as "Other" for simplicity. The full-table scan threshold is reached for most values, although more are possible compared to a boolean type. It is important to know how values are distributed in reality and not only how many distinct values are possible - a property that is often not considered when benchmarking an approach.

Figure A

Boolean-Like Distribution of Enum Values



This issue also appears in many more columns. For example, an issue tracker will store the current state of each software bug. This time, an unfavorable value distribution will appear after a longer time because the number of closed issues will far exceed the open ones after a few months or years.

> **Info**
>
> Boolean columns are always a bad fit for indexes: Their property of having only two values usually leads to them having unfavourable values distribution making an index useless. You should only index them in rare cases when searching for an uncommon value only used for a low single-digit percentage of the rows.

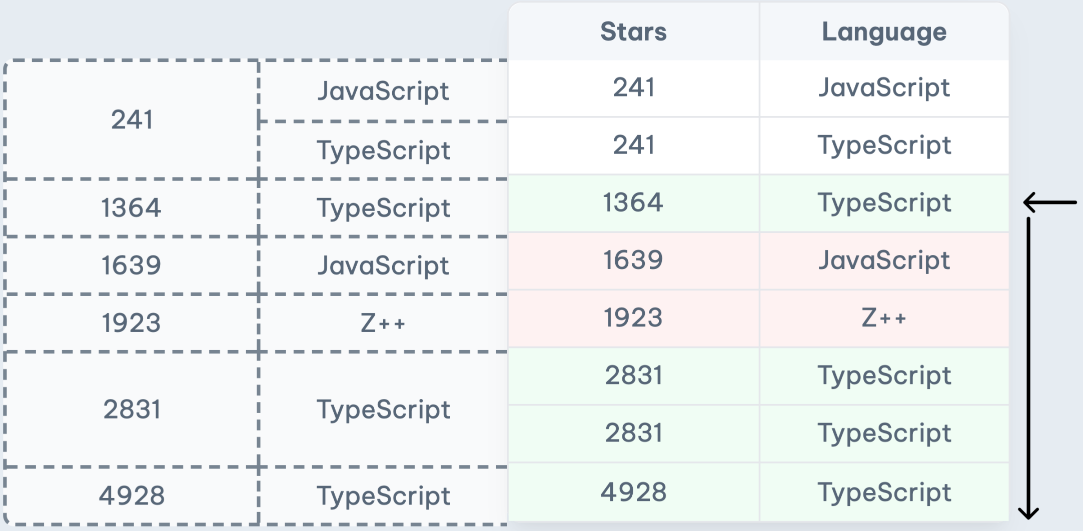
5.3 Transforming Range Conditions

Queries involving range conditions (e.g. `WHERE stars > 1000`) additionally to other filters or index operations are problematic. Anytime a range condition is used, the index columns right to that condition can not narrow the index entries to check (*Index Access Principle 4: Scan On Range Conditions*).

Let's take a closer look at this on a simple query searching for popular TypeScript repositories on GitHub defined by a large stars count (`WHERE language = 'TypeScript' AND stars > 1000`) and an index on the `stars` and `language` columns (Fig. A). The entire range for the `stars` condition is scanned and the index entries are filtered by the `language` constraint.

Figure A

Index on **(stars, language)**



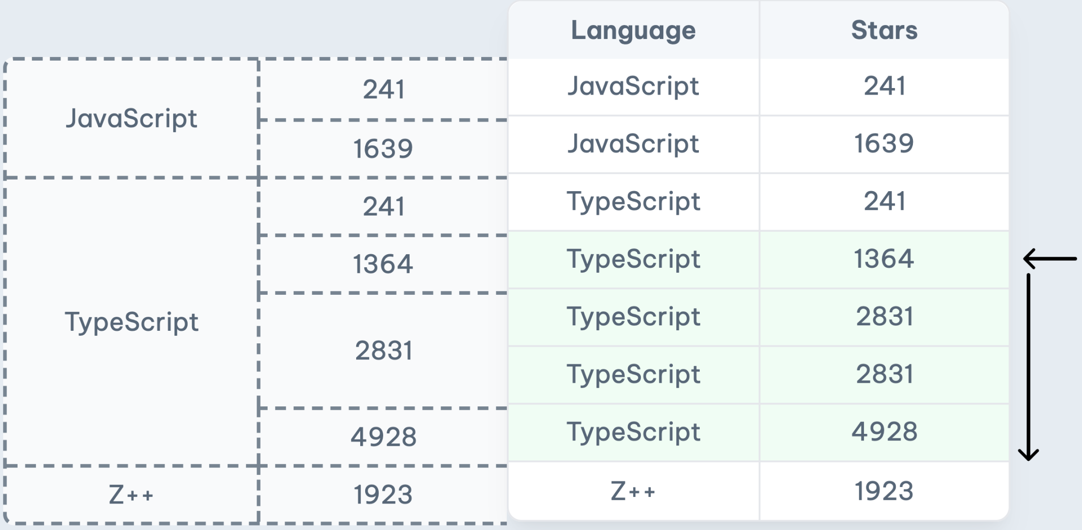
Stars	Language
241	JavaScript
241	TypeScript
1364	TypeScript
1639	JavaScript
1923	Z++
2831	TypeScript
2831	TypeScript
4928	TypeScript

Moving the `stars` column last makes the index more efficient because the index records can be first filtered on the `language` condition (Fig. B). Due to the misplaced range column, more records than necessary had to be evaluated in the previous index.

Solving the problem for simple filter conditions is easy. However, more than sorting columns is required if a query has several range conditions. Furthermore, you want to apply more operations to the index than filtering.

Figure B

Index on **(language, stars)**

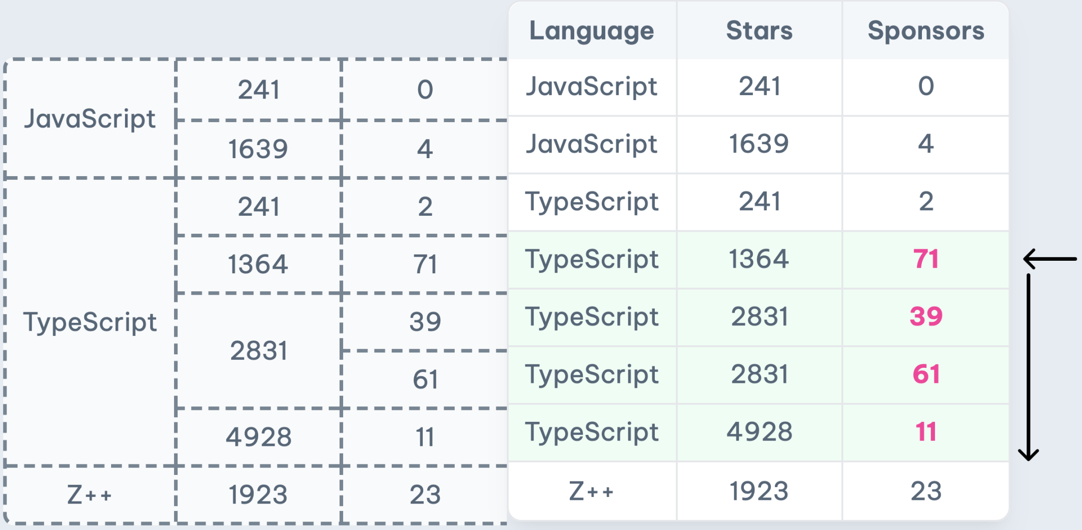


Language	Stars
JavaScript	241
JavaScript	1639
TypeScript	241
TypeScript	1364
TypeScript	2831
TypeScript	2831
TypeScript	4928
Z++	1923

Let's extend the query by also sorting on the number of sponsors the repository has (`WHERE language = 'TypeScript' AND stars > 1000 ORDER BY sponsors ASC`). The values in the index (Fig. C) are sorted in ascending order for each star count and not in a consecutive way to get them in the sorting order. Thus, an additional sorting step is needed after selecting the matching index entries. You should carefully consider whether an index fits a query if range conditions are used.

Figure C

Index on **(language, stars, sponsors)**



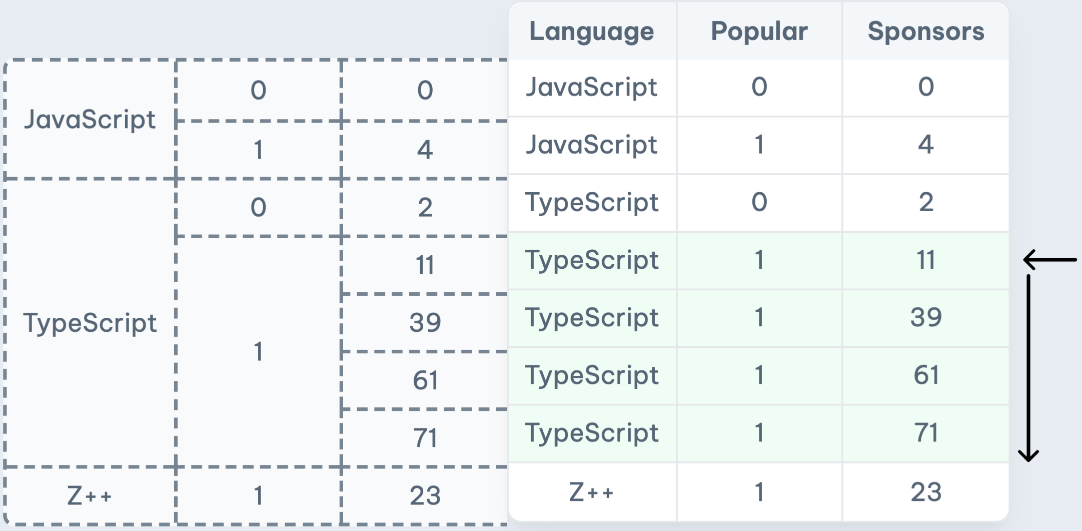
Language	Stars	Sponsors
JavaScript	241	0
JavaScript	1639	4
TypeScript	241	2
TypeScript	1364	**71**
TypeScript	2831	**39**
TypeScript	2831	**61**
TypeScript	4928	**11**
Z++	1923	23

Virtual Columns

One way to optimize a query with range conditions is to introduce virtual columns. The idea is to remove the problematic behavior of range conditions that have to scan over many index entries and can't use columns in an index on the right anymore. The range condition is transformed into a `popularity` value that indicates whether the condition matches (1) or not (0) and can be used with an equality condition (Fig. D). The filtering and sorting can now be applied to the index as the values are optimally arranged.

Figure D

Index on **(language, popular, sponsors)**



Language	Popular	Sponsors
JavaScript	0	0
JavaScript	1	4
TypeScript	0	2
TypeScript	1	11
TypeScript	1	39
TypeScript	1	61
TypeScript	1	71
Z++	1	23

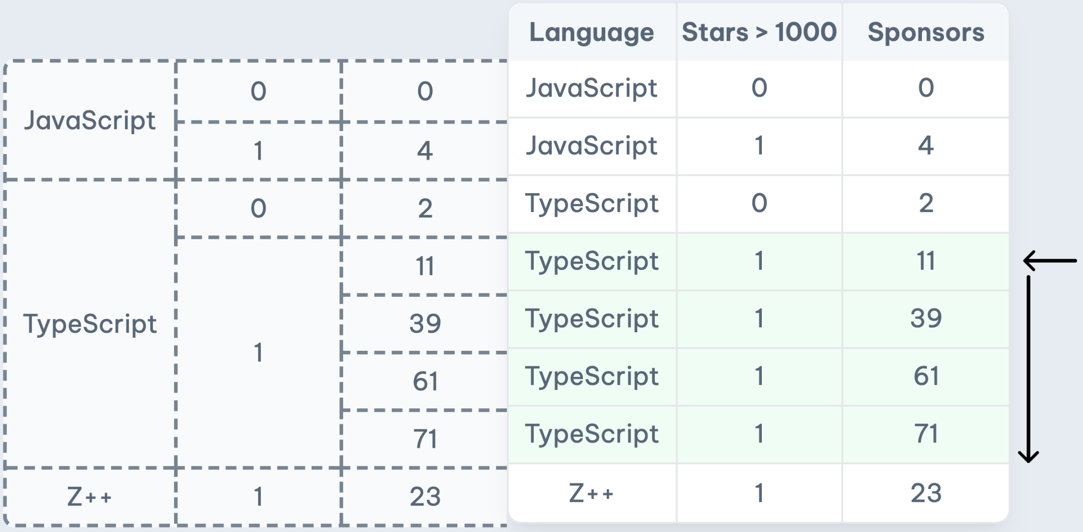
Unfortunately, not every database supports virtual columns (e.g. PostgreSQL). You can use stored generated columns, but they require more space: The value must be stored in the table and index.

Functional Indexes

Those who do not like columns in tables for this purpose can also use functional indexes. The idea is to execute the transformation directly in the query and create a suitable index. For example, with MySQL the short form of the condition can be declared as `IF(stars > 1000, 1, 0)`. The query would then be `WHERE language = 'TypeScript' AND IF(stars > 1000, 1, 0) = 1 ORDER BY sponsors ASC` and a suitable index would contain the corresponding transformation (Fig. E).

Figure E

Index on **(language, IF(stars > 1000, 1, 0), sponsors)**



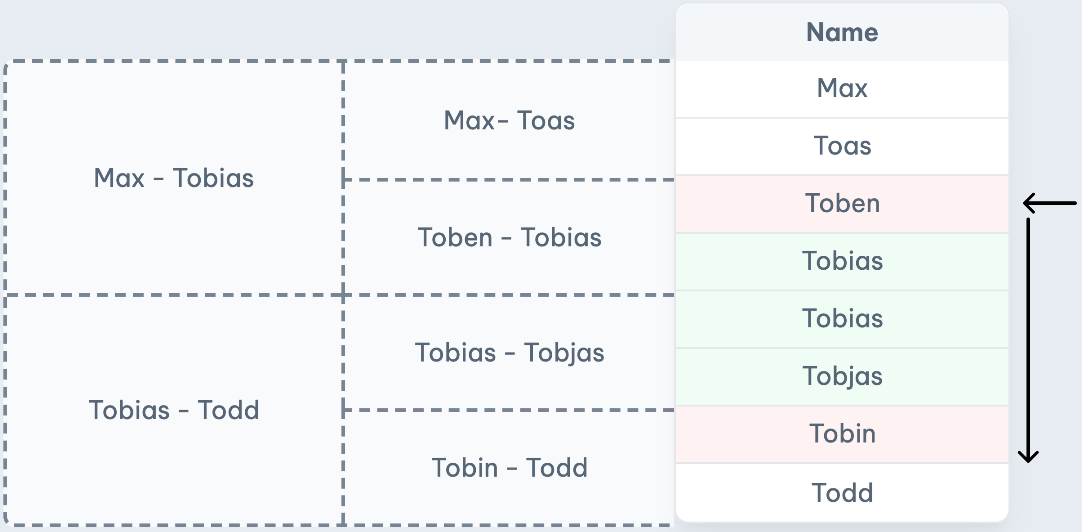
Language	Stars > 1000	Sponsors
JavaScript	0	0
JavaScript	1	4
TypeScript	0	2
TypeScript	1	11
TypeScript	1	39
TypeScript	1	61
TypeScript	1	71
Z++	1	23

5.4 Leading Wildcard Search

The chapter "*3.3 Pattern Matching*" explained the indexing strategy for pattern matching with wildcards: A pattern matching condition like `WHERE name LIKE 'Tob%s'` is transformed to a range condition (`'WHERE name >= 'Tob' AND name < 'Toc'`) to scan for any value following `Tob` (the first possible match) to anything before `Toc` (the first match that wouldn't fit anymore). Those index entries are checked to see whether they match the pattern (Fig. A).

Figure A

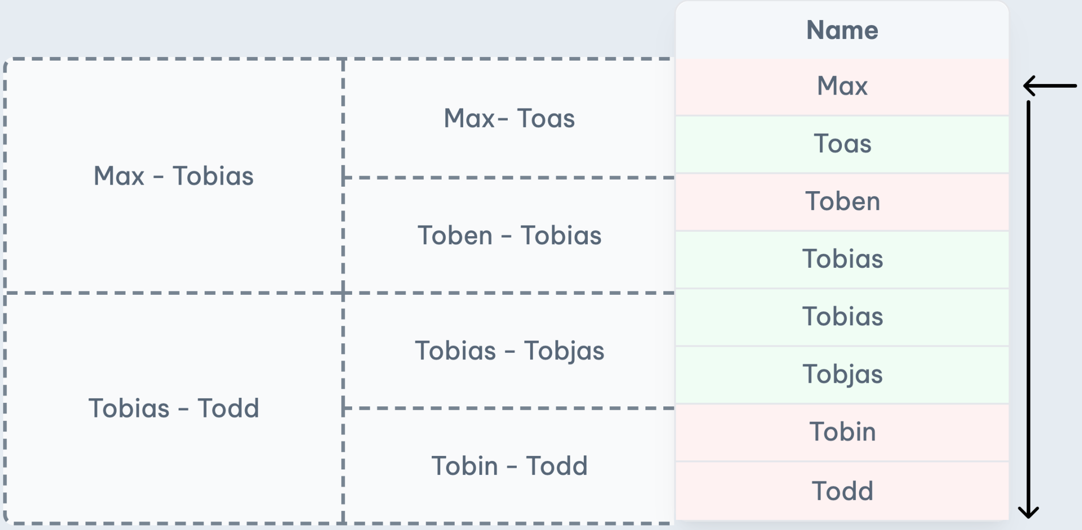
Index on **(name)**



Conditions using a wildcard as the first character for the pattern (called leading wildcards) are problematic. You would expect the query to be executed as shown in Fig. B, but this is different from what is happening: The database cannot estimate how many index entries will match a pattern with leading wildcards and therefore can not calculate the cost to load these matching rows. An index will only be used when it is possible to evaluate whether using it would be fast or slow! Therefore, the database will fall back to using a full-table scan. For this reason, you should always avoid leading wildcards.

Figure B

Index on **(name)**



Unfortunately, creating an index for this special case is impossible. However, there is one exception: PostgreSQL is the only database with a unique index type for this. It ships with a standardly available extension (pg_trgm) to create trigram indexes that allow wildcards at any position.

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;

CREATE INDEX trgm_idx ON contacts USING GIN (name gin_trgm_ops);
```

The way this index works is very original: Words are fragmented into all possible three-character long strings (trigram). For example, 'Tobias Petry' is split into the lowercase trigrams 'tob', 'obi', 'bia', 'ias', 'as ', 's p', ' pe', 'pet', 'etr', 'try'.

Such a trigram index can now be used if the search pattern contains at least three consecutive characters without a wildcard. However, these indexes can become very large due to the vast number of trigrams.

5.5 Type Juggling

Like programming languages, databases can be loosely typed, strictly typed, or somewhere in between. This decision has far-reaching consequences. A loosely typed database is easier to work with as you can be inaccurate with types but it also has to do a lot of implicit type conversions. Some of them are totally fine and some are just weird. One of those odd issues is a common problem affecting anyone using MySQL.

```sql
SELECT *
FROM orders
WHERE payment_id = 57013925718
```

The former query searching for a specific order by a payment provider's ID looks fine and should work fast when an index has been created on the column. However, the person making the schema didn't know whether the payment provider's IDs were integers or character sequences. So they used a `varchar(255)` column.

The problem with this query is matching two incompatible types to compare them (type juggling): The schema uses a `varchar(255)` column but the query uses an `int` comparison value. One would expect the integer to be converted to a string, as this is the most sensible way to compare them. But in MySQL, a string is always converted to an integer in these cases.

```sql
SELECT *
FROM orders
WHERE CAST(payment_id AS UNSIGNED) = 57013925718
```

Therefore, the index can no longer be used because the column is transformed - as explained in chapter *“4.1 The Index Can’t Be Used”*. This problem often appears with older applications when the schema is not updated to reflect refactorings or business changes. You should remember this problem whenever you see a text column storing numbers.

The opposite case with `WHERE int = varchar(255)` does not lead to a problem: By converting strings to integers, the transformation only happens on the value and not the column (`WHERE int = CAST(varchar(255) AS UNSIGNED)`).

5.6 Index-Only Queries

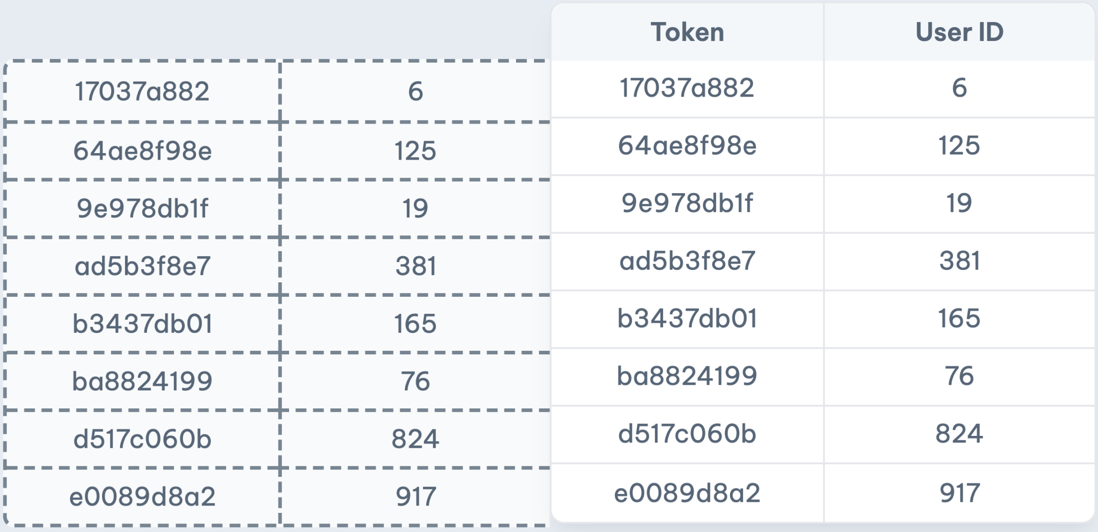
Another simple and powerful performance optimization has not been discussed so far. All table columns of rows an index identified are loaded from the disk (“*1.2 The Interaction of Indexes and Tables*”). But sometimes we only need precisely one or two columns and the remaining ones are not needed. The effort to load all of them is unnecessary.

```sql
SELECT user_id FROM sessions WHERE token = 'b3437db01';
```

Queries like this are present in all applications. They are usually simple lookup queries that map one piece of information (e.g. email address, country) to another (e.g. user id, tax rate). Such queries can be optimized to read the mapping information from the index and not the table's row (Fig. A).

Figure A

Index on **(token, user_id)**



17037a882	6
64ae8f98e	125
9e978db1f	19
ad5b3f8e7	381
b3437db01	165
ba8824199	76
d517c060b	824
e0089d8a2	917

Token	User ID
17037a882	6
64ae8f98e	125
9e978db1f	19
ad5b3f8e7	381
b3437db01	165
ba8824199	76
d517c060b	824
e0089d8a2	917

Any columns added to the end of an index can be used to directly retrieve those values without loading the referenced row. This may not sound like a great feature on its own at first. However, the performance of such queries is much better than those that must load table rows - there is one action less to do. This is called an index-only query as it can be completed entirely from the index without touching any table row. But you should use this optimization carefully as the storage size increases with each column added to an index.

It is recommended to use them for simple but frequently executed queries like the authentication check or mapping one piece of information to another. By far, the best usage is for m:n tables connecting two tables with a join: They are constantly searched for an ID, and another ID is used from the row to look up rows in the next table. Creating indexes for these tables with this optimization is always a good idea.

5.7 Filtering and Sorting With Joins

Tables are always structured according to logical boundaries and dictate how data is separated. This often reflects the object-oriented data model but only partially fits the querying needs. Again and again, several tables have to be linked to any data. This doesn't seem like a problem as joins are designed for this and (hopefully) perfect indexes have been created. But every join iteration slows down your query a little bit...

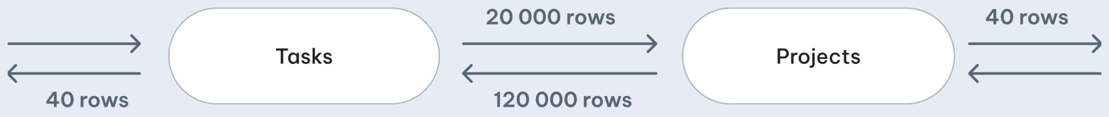
Let's imagine a ToDo application that organizes tasks into individual projects. As usual, all open tasks should be displayed on the dashboard. However, when a project is marked as completed, its tasks should no longer appear.

```sql
SELECT tasks.*
FROM tasks
JOIN projects USING(project_id)
WHERE tasks.team_id = 4 AND taks.status = 'open' AND projects.status = 'open';
```

Executing this query on your test data will always be fast. However, the query is slow for big teams with thousands of tasks and hundreds of projects in production with a lot more data. The problem is the table structure rather than the use of joins.

Figure A

Workflow of Different Join Orders



The real problem is both tables alone could not filter the rows significantly and need to join many times to calculate the result (Fig. B): Starting with the tasks table would match about 20,000 rows for the team, which will be filtered to just 40 results by joining the projects table and removing all tasks for closed projects. Alternatively, 120,000 open projects are reduced to the same result by joining the tasks table and only keeping the rows for the specific team.

With both approaches, thousands of join iterations are needed to get the intended result and each one costs time. This query can not be improved with better indexes. The schema must be changed to enable all filtering on just one table instead of needing both. The most straightforward approach is marking tasks as done when the project is no longer active or storing a copy of the project status in the tasks table. With both approaches, the join is no longer needed.

An identical problem exists with the following query: The filtering is executed on one table and the sorting on another. Again, many rows must be joined and then sorted afterward, only to throw away the majority and keep 30 of them.

```sql
SELECT *
FROM invoices
JOIN invoices_metadata USING(invoice_id)
WHERE invoices.tenant_id = 4236
ORDER BY invoices_metadata.due_date
LIMIT 30
```

> **Tip**
>
> You should not limit your focus to the correct indexes for join conditions. A query that has to execute tens or hundreds of thousands of join loops will always take a long time.

5.8 Exceeding the Maximum Index Size

An index cannot be infinitely large. It must be significantly smaller than the table row to fit into memory and thereby make a query faster. Consequently, you will get an error message like "*The index row size of 3480 bytes exceeds the maximum size of 2712 bytes for index contacts_fullname*"' if you try adding more and more columns at some point. The limit on how big an index can be varies depending on the database. You won't reach the threshold when adding dozens of number columns, but you will run into it for many columns storing big strings.

Prefix Index

A simple approach is to limit the length of long text columns. Do all possible 1000 characters of a blog title need to be indexed when a typical maximum length is much smaller? In most cases, significantly fewer characters are sufficient.

```
CREATE INDEX articles_search ON articles (type, (substring(title, 1,
20)));

SELECT *
FROM articles
WHERE substring(title, 1, 20) = '...20 chars truncated title...' AND
title = '...full title...'
```

The idea of prefix indexes is to index only a tiny part of the whole column to save storage by indexing only the first characters. Such an index is utilized using a condition on the same truncation as the index definition (`WHERE substring(title, 1, 20) = ?`). But the query's result can include unwanted rows with identical truncation but different actual column values. So, another condition has to be used to reject these incorrect results by comparing the entire string (`WHERE title = '...full title...'`). The index finds prefix matches that will be filtered again when the whole string has been loaded from the table's row.

With MySQL, this concept is easier to implement because the optimization is built-in: The prefix length can be defined directly in the index definition for all large data types (e.g. `varchar`, `text`, `blob`). Writing the query is also easier because the normal condition can be used directly and MySQL takes care of all the needed logic for prefix-matching.

```
CREATE INDEX articles_search ON articles (type, title(20));

SELECT * FROM articles WHERE title = '...full title...'
```

With prefix indexes, you must always find a balance between the length of the prefix and incorrect matches: If the prefix is too short, many rows must be loaded from the table to check again the entire value. If the prefix is too long, the index is unnecessarily bloated and requires more memory than needed.

Indexing Hash Values

Another approach is to store a hash value (e.g. md5, sha1) of the long content. Compared to a prefix index, there is (almost) never a collision for the same hash value. So, there aren't many table rows loaded and removed when the entire string is checked. The efficiency is increased!

```
CREATE INDEX articles_search ON articles (type, (sha1(title)));

SELECT *
FROM articles
WHERE sha1(title) = '...hash of title...' AND title = '...full title...'
```

In some cases, the first characters of a prefix index are often identical but the increased space required by hash values is undesirable. However, you do not have to choose one of the two solutions as you can combine both: A hash value on the entire string will be completely different and has uniformly distributed values even if multiple strings share a common long prefix. This hash value can then be stored with the prefix approach that limits it to a smaller size.

```
CREATE INDEX articles_search ON articles (type, (SUBSTRING(sha1(title),
1, 20)));

SELECT *
FROM articles
WHERE SUBSTRING(sha1(title), 1, 20)) = '...shortened hash of title...'
AND title = '...full title...'
```

You shouldn't store those hash values as text columns as this will take a lot of storage. The best approach is to store them in binary format - the method for this depends on your database.

You can easily optimize the size using a database like PostgreSQL or MariaDB that provides storage-efficient UUID columns. The values of md5 hashes and UUIDs are interchangeable because they are both 128-bit long. Below is an example for PostgreSQL in which a md5 value is calculated and converted to a storage-efficient UUID.

```
CREATE INDEX articles_search ON articles (type, (md5(title)::uuid));

SELECT *
FROM articles
WHERE md5(title)::uuid = md5('...full title...')::uuid AND title =
'...full title...'
```

Multiple Columns

Another possible optimization applies if your index includes several columns and is still within the index size limit: The space requirement of several columns can be reduced by hashing them together as a concatenated value. This can be beneficial for massive tables or database servers with little memory.

5.9 JSON Objects and Arrays

A few years ago, NoSQL schemaless databases were widely promoted to replace relational databases like MySQL or PostgreSQL completely. This hype significantly influenced the development of databases: Everyone now has a JSON data type to store schema-less data.

Unfortunately, indexing JSON values is still not easy: An index on a JSON column can only be used to search for rows with an identical JSON value but not for specific values within it. As this is not very helpful, you can create specialized indexes for JSON objects and arrays - but these are very different depending on the database used. Therefore, only MySQL and PostgreSQL will be discussed in the following sections.

JSON Objects

It is impossible to create indexes for JSON objects that cover all keys - except for PostgreSQL GIN indexes explained last. You must choose one of the following two approaches for every JSON key that should be indexed.

Virtual Columns

The easiest way is to create virtual columns that extract a JSON value and are indexed like any regular column. The example below shows the syntax for MySQL.

```
CREATE TABLE contacts (
  id bigint PRIMARY KEY AUTO_INCREMENT,
  attributes json NOT NULL,
  email varchar(255)  AS (attributes->>"$.email") VIRTUAL NOT NULL,
  INDEX contacts_email (email)
);

SELECT * FROM contacts WHERE attributes->>"$.email" =
'admin@example.com';
```

Warning

Strings are extracted in MySQL from a JSON object with the collation utf8mb4_bin. Therefore, values are compared case-sensitively unlike the usual behavior of comparing them case-insensitive!

This approach is straightforward, but not everyone will like it: The table becomes more confusing for every additional JSON attribute stored as a virtual column. Furthermore, the approach takes more storage space if you use a database that doesn't support virtual columns (e.g. PostgreSQL) because the JSON value has to be stored in the column and the index.

Functional Indexes

You can also use functional indexes instead of virtual columns - the cleanest solution. As explained in “*5.1 Indexes on Functions*”, you only have to create the index on the JSON operation and the matching will automatically be used. The following is an example for Postgresql.

```
CREATE INDEX contacts_email ON contacts ((attributes->>'email'));

SELECT * FROM contacts WHERE attributes->>'email' = 'admin@example.com';
```

But functional indexes on JSONs are more complicated with MySQL: Trying to create an index similarly will cause an error (“*Cannot create a functional index on an expression that returns a BLOB or TEXT*”). The value must first be cast to a string and then changed to the collation used by MySQL for JSON values (`utf8_mb4_bin`). MySQL can only use the index for queries if you follow all these steps.

```
CREATE INDEX contacts_email ON contacts ((
  CAST(attributes->>"$.email" AS CHAR(255)) COLLATE utf8mb4_bin
));

SELECT * FROM contacts WHERE attributes->>"$.email" =
'admin@example.com';
```

GIN Indexes

However, creating an index for every JSON key is very inconvenient. PostgreSQL provides the GIN index type (Generalized Inverted Index) that can index an entire JSON object with all values it contains. You no longer have to create multiple functional indexes!

```
CREATE INDEX contacts_attributes ON contacts USING gin (attributes);
```

But they come with a minor drawback: You must use specialized comparison operators instead of the standard ones (e.g. `=` or `>`). However, these operators are also beneficial as they can do different remarkable things:

- `json @> json`: The left JSON value must contain the right JSON value.
- `json ? text`: The text value must be present as a JSON key.
- `json ?| array`: Any of the array values must be present as JSON keys.
- `json ?& array`: All of the array values must be present as JSON keys.
- `json @? jsonpath`: The JSON path has to match at least one element.

```
SELECT * FROM contacts WHERE attributes @> '{"email":
"admin@example.com"}';
SELECT * FROM contacts WHERE attributes ? 'email';
SELECT * FROM contacts WHERE attributes ?| array['email', 'phone'];
SELECT * FROM contacts WHERE attributes ?& array['email', 'phone'];
SELECT * FROM contacts WHERE attributes @? '$.tags[*] ? (@.type ==
"note" && @.severity > 3)';
```

JSON Arrays

In addition to JSON objects, you may also be interested in indexing JSON arrays. This is a powerful but less-known feature. You can e.g. save the categories a product belongs to into a JSON array to check whether a product belongs to two categories but not a specific other one.

With PostgreSQL, you can use the same capabilities as described for JSON Objects to check for overlapping JSON values:

```
CREATE INDEX products_categories ON products USING gin (categories);

SELECT * FROM products WHERE categories @> '["printed book", "ebook"]'
AND NOT categories @> '["audiobook"]';
```

For MySQL, you can use so-called multi-valued indexes that store multiple values for a given column. However, this index type is limited to unsigned integers and the following example therefore uses the category IDs instead of their names.

```
CREATE INDEX products_categories on products ((CAST(categories AS
UNSIGNED ARRAY)));

SELECT * FROM contacts WHERE JSON_CONTAINS(attributes, CAST('[17, 23]'
AS JSON))  AND NOT JSON_CONTAINS(attributes, CAST('[11]' AS JSON))
```

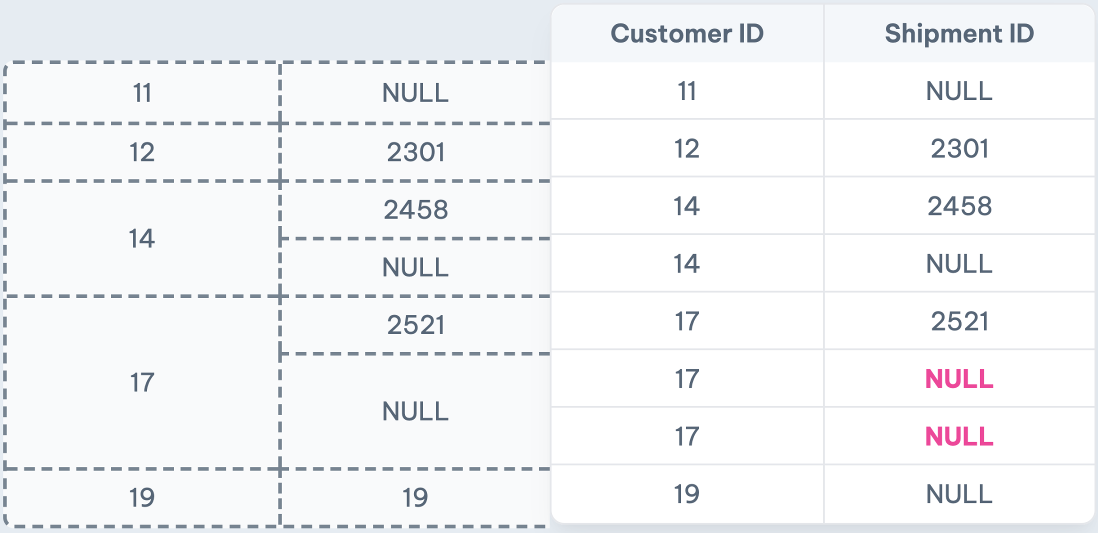
5.10 Unique Indexes and Null

NULL values have special behavior in SQL that they are not comparable to anything. So, any NULL value is different from another NULL value. This sounds highly theoretical but severely influences unique keys not behaving like you would expect them to.

Imagine a shop application where customers can order highly anticipated products - e.g., a new game console or smartphone. To guarantee equal opportunities for everyone, customers can only have one open order for these products at any time. An easy solution would be to create a unique index on the columns `customer_id` and `shipment_id` that is `NULL` when the customer is still waiting and will be replaced as soon as the order is shipped. However, the index is different than expected due to the NULL behavior (Fig. A).

Figure A

Index on **(customer_id, shipment_id)**



Customer ID	Shipment ID
11	NULL
12	2301
14	2458
14	NULL
17	2521
17	**NULL**
17	**NULL**
19	NULL

Both NULL values for customer 17 are claimed to be not identical as a NULL value can't be compared to another NULL value. For this reason, several duplicate entries can exist in a unique index - a circumstance that one would not expect. You should always look out for unique indexes when they include nullable columns.

Transformations

An easy way to ensure the uniqueness of entries in unique indexes is to transform the null values to non-null values. You can achieve this by creating a functional index applying the transformation. Consequently, only comparable values exist in the index and the uniqueness guarantee is satisfied again.

But keep in mind that the `shipment_id` column has been transformed. A query using a condition on both columns can't use this index anymore. You have to create another one to support searching the data.

```sql
CREATE UNIQUE INDEX uniqueness_idx ON orders (
  customer_id,
  (CASE WHEN shipment_id IS NULL THEN -1 ELSE shipment_id END)
);
```

NULLS NOT DISTINCT

The SQL standard suggests an elegant approach to solve the problem. You can tell the database to treat all null values as identical for the unique index. However, this feature is currently only implemented by PostgreSQL. This may change over time and more databases will support it.

```sql
CREATE UNIQUE INDEX uniqueness_idx ON orders (
  customer_id, shipment_id
) NULLS NOT DISTINCT;
```

5.11 Location-Based Searching With Bounding-Boxes

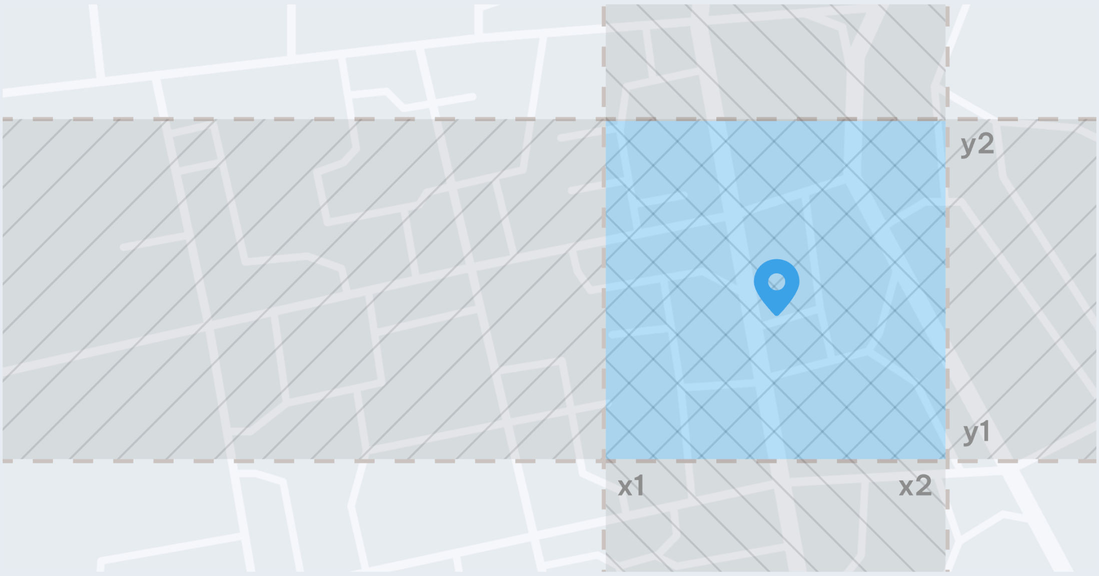
Many applications use location-based search for certain functionalities such as finding restaurants in the local area to eat out today. These proximity searches are easy to implement in SQL. A bounding box defined by the longitude (x-axis) and the latitude (y-axis) is used to express a range to search within (Fig. A).

```
CREATE TABLE businesses (
  id bigint PRIMARY KEY NOT NULL,
  type varchar(255) NOT NULL,
  name varchar(255) NOT NULL,
  latitude float NOT NULL,
  longitude float NOT NULL
);
CREATE INDEX search_idx ON businesses (longitude, latitude);

SELECT *
FROM businesses
WHERE type = 'restaurant' longitude BETWEEN -73.9752 AND -74.0083 AND
latitude BETWEEN 40.7216 AND 40.7422
```

Figure A

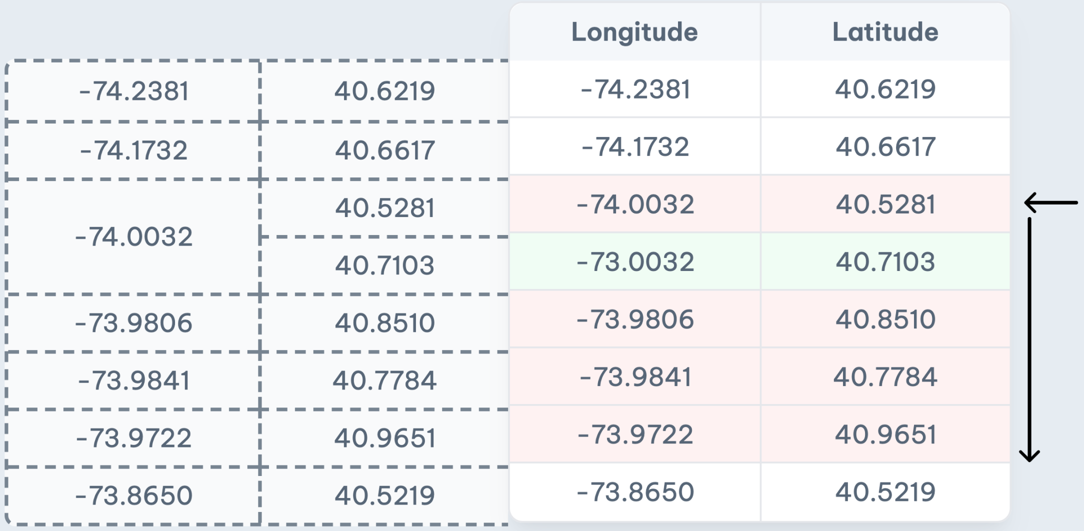
Bounding-Box to Define Search Area



Even though this is the most used code, it could be better. The query is problematic because two range conditions are used: Only the condition for the longitude can limit the index records as an index scan is triggered on the first range condition (*Index Access Principle 4: Scan On Range Conditions*). Therefore, all index records for the range `longitude BETWEEN -73.9752 AND -74.0083` are validated whether the latitude condition matches (Fig. B). This is fine for small databases but not for larger ones storing entries for a whole country or the world. The large number of index entries matching the longitude range but not the latitude one will have severe performance impacts.

Figure B

Index on **(longitude, latitude)**



-74.2381	40.6219
-74.1732	40.6617
-74.0032	40.5281
	40.7103
-73.9806	40.8510
-73.9841	40.7784
-73.9722	40.9651
-73.8650	40.5219

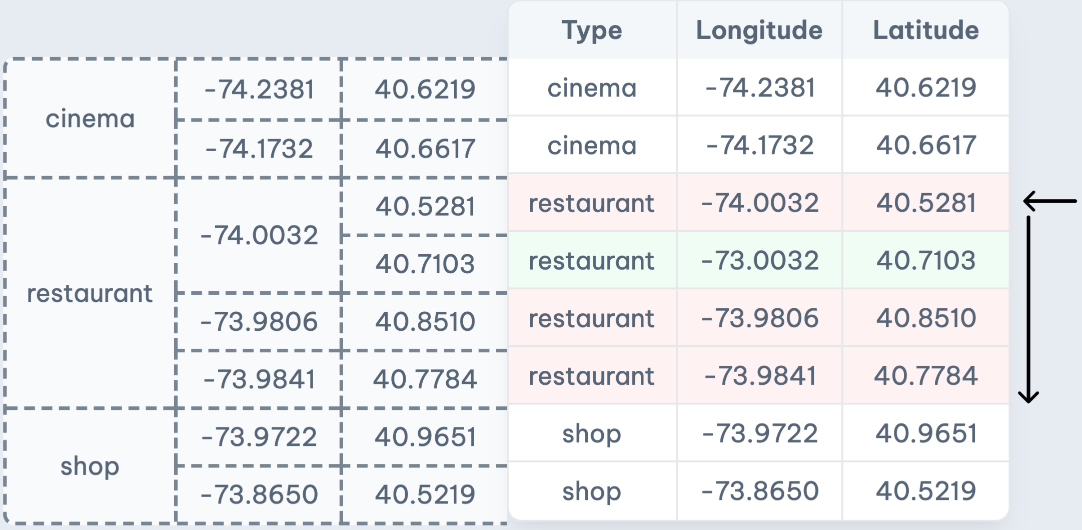
Longitude	Latitude
-74.2381	40.6219
-74.1732	40.6617
-74.0032	40.5281
-73.0032	40.7103
-73.9806	40.8510
-73.9841	40.7784
-73.9722	40.9651
-73.8650	40.5219

Additional Columns

A quick optimization method is to make the index more fitting. The condition on the type column was previously not part of the index but can limit the amount of index records to scan (Fig. C). However, the range scan problem has only improved a little in efficiency and has not been solved. The number of index records scanned is still too high but the performance could probably be sufficient for you now.

Figure C

Index on **(type, longitude, latitude)**



Type	Longitude	Latitude
cinema	-74.2381	40.6219
cinema	-74.1732	40.6617
restaurant	-74.0032	40.5281
restaurant	-73.0032	40.7103
restaurant	-73.9806	40.8510
restaurant	-73.9841	40.7784
shop	-73.9722	40.9651
shop	-73.8650	40.5219

Spatial Indexes

Location-based searching cannot be executed efficiently with regular indexes. The developers of databases realized this long ago and created specialized indexes: Spatial Indexes.

This index type is designed to make points or more complex shapes efficiently searchable in a multidimensional space (e.g. the three dimensions on earth). Using them is as easy as using any other index. The following example shows the optimized example for PostgreSQL. The table must be changed to store a point in spatial reference 4326 instead of separate latitude and longitude values. This reference defines how geometry is referenced to locations on earth's surface and is used to respect the earth's curvature in calculations. More reference systems exist for more precise positioning - e.g., specific to countries or regions.

```sql
CREATE TABLE businesses (
  id bigint PRIMARY KEY NOT NULL,
  type varchar(255) NOT NULL,
  name varchar(255) NOT NULL,
  location geometry(point, 4326) NOT NULL
);
CREATE INDEX search_idx ON boundingbox USING GIST (type, location);
```

The query must only be rewritten slightly to use spatial search functions to check whether any stored point is within the bounding box.

```
SELECT *
FROM boundingbox
WHERE type = 'restaurant' and location && ST_MakeEnvelope (
  /* longitude_min */ -73.9752, /* latitude_min */ 40.7216,
  /* longitude_max */ -74.0083, /* latitude_max */ 40.7422,
  /* spatial reference*/ 4326
)
```

The same optimization can be executed for MySQL, as shown with the following code. But MySQL has some differences in its spatial implementation:

- A spatial index can only have one column. So prefixing the index by the type column to segment the spatial index will not work.
- The global position reference (srid=4326) is not supported by all functions, e.g. by the used `ST_MakeEnvelope`. Its not a problem for this purpose as we only need entries within the bounding box but distance calculations would be slightly off because earth's curvature will be ignored.

```
CREATE TABLE businesses (
  id bigint PRIMARY KEY NOT NULL,
  type varchar(255) NOT NULL,
  name varchar(255) NOT NULL,
  location point SRID 0 NOT NULL,
);
CREATE SPATIAL INDEX search_idx ON businesses (location);

SELECT *
FROM businesses
WHERE type = 'restaurant' and ST_CONTAINS(
  ST_MakeEnvelope(
    point(/* longitude_min */ -73.9752, /* latitude_min */ 40.7216),
    point(/* longitude_max */ -74.0083, /* latitude_max */ 40.7422)
  ),
  location
);
```