



Database index

From Wikipedia, the free encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
[العربية](#) [Deutsch](#) [Español](#) [Français](#) [한국어](#) [日本語](#) [Português](#) [Русский](#) [中文](#)

15 more

A **database index** is a [data structure](#) that improves the speed of data retrieval operations on a [database table](#) at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more [columns of a database table](#), providing the basis for both rapid random [lookups](#) and efficient access of ordered records.

An index is a copy of selected columns of data from a table, called a [database key](#) or simply [key](#), that can be searched very efficiently that also includes a low-level disk block address or direct link to the complete row of data it was copied from. Some databases extend the power of indexing by letting developers create indexes on functions or [expressions](#). For example, an index could be created on `upper(last_name)`, which would only store the upper-case versions of the `last_name` field in the index. Another option sometimes supported is the use of [partial indices](#), where index entries are created only for those records that satisfy some conditional expression. A further aspect of flexibility is to permit indexing on [user-defined functions](#), as well as expressions formed from an assortment of built-in functions.

Contents [hide]

1	Usage
1.1	Support for fast lookup
1.2	Policing the database constraints
2	Index architecture and indexing methods
2.1	Non-clustered
2.2	Clustered
2.3	Cluster
3	Column order
4	Applications and limitations
5	Types of indexes
5.1	Bitmap index
5.2	Dense index
5.3	Sparse index
5.4	Reverse index
5.5	Primary index
5.6	Secondary index
6	Index implementations
6.1	Index concurrency control
7	Covering index
8	Standardization
9	See also
10	References

Usage [edit]

Support for fast lookup [edit]

Most [database](#) software includes indexing technology that enables [sub-linear time lookup](#) to improve performance, as [linear search](#) is inefficient for large databases.

Suppose a database contains N data items and one must be retrieved based on the value of one of the fields. A simple implementation retrieves and examines each item according to the test. If there is only one matching item, this can stop when it finds that single item, but if there are multiple matches, it must test everything. This means that the number of operations in the worst case is O(N) or [linear time](#). Since databases may contain many objects, and since lookup is a common operation, it is often desirable to improve performance.

An index is any data structure that improves the performance of lookup. There are many different [data structures](#) used for this purpose. There are complex design trade-offs involving lookup performance, index size, and index-update performance. Many index designs exhibit logarithmic ($O(\log N)$) lookup performance and in some applications it is possible to achieve flat ($O(1)$) performance.

Policing the database constraints [edit]

Indexes are used to police [database constraints](#), such as [UNIQUE](#), [EXCLUSION](#), [PRIMARY KEY](#) and [FOREIGN KEY](#). An index may be declared as [UNIQUE](#), which creates an implicit constraint on the underlying table. Database systems usually implicitly create an index on a set of columns declared [PRIMARY KEY](#), and some are capable of using an already-existing index to police this constraint. Many database systems require that both referencing and referenced sets of columns in a [FOREIGN KEY](#) constraint are indexed, thus improving performance of inserts, updates and deletes to the tables participating in the constraint.

Some database systems support an [EXCLUSION](#) constraint that ensures that, for a newly inserted or updated record, a certain predicate holds for no other record. This can be used to implement a [UNIQUE](#) constraint (with equality predicate) or more complex constraints, like ensuring that no overlapping time ranges or no intersecting geometry objects would be stored in the table. An index supporting fast searching for records satisfying the predicate is required to police such a constraint.^[1]

Index architecture and indexing methods [edit]

Non-clustered [edit]

The data is present in arbitrary order, but the [logical ordering](#) is specified by the index. The data rows may be spread throughout the table regardless of the value of the indexed column or expression. The non-clustered index tree contains the index keys in sorted order, with the leaf level of the index containing the pointer to the record (page and the row number in the data page in page-organized engines; row offset in file-organized engines).

In a non-clustered index,

- The physical order of the rows is not the same as the index order.
- The indexed columns are typically non-primary key columns used in JOIN, WHERE, and ORDER BY clauses.

There can be more than one non-clustered index on a database table.

Clustered [edit]

Clustering alters the data block into a certain distinct order to match the index, resulting in the row data being stored in order. Therefore, only one clustered index can be created on a given database table. Clustered indices can greatly increase overall speed of retrieval, but usually only where the data is accessed sequentially in the same or reverse order of the clustered index, or when a range of items is selected.

Since the physical records are in this sort order on disk, the next row item in the sequence is immediately before or after the last one, and so fewer data block reads are required. The primary feature of a clustered index is therefore the ordering of the physical data rows in accordance with the index blocks that point to them. Some databases separate the data and index blocks into separate files, others put two completely different data blocks within the same physical file(s).

Cluster [edit]

When multiple databases and multiple tables are joined, it's referred to as a [cluster](#) (not to be confused with clustered index described above). The records for the tables sharing the value of a cluster

key shall be stored together in the same or nearby data blocks. This may improve the joins of these tables on the cluster key, since the matching records are stored together and less I/O is required to locate them.^[2] The cluster configuration defines the data layout in the tables that are parts of the cluster. A cluster can be keyed with a [B-Tree](#) Index or a [hash table](#). The data block where the table record is stored is defined by the value of the cluster key.

Column order [\[edit\]](#)

The order that the index definition defines the columns is important. It is possible to retrieve a set of row identifiers using only the first indexed column. However, it is not possible or efficient (on most databases) to retrieve the set of row identifiers using only the second or greater indexed column.

For example, imagine a phone book that is organized by city first, then by last name, and then by first name. If you are given the city, you can easily extract the list of all phone numbers for that city. However, in this phone book it would be very tedious to find all the phone numbers for a given last name. You would have to look within each city's section for the entries with that last name. Some databases can do this, others just won't use the index.

In the phone book example with a [composite index](#) created on the columns (`city, last_name, first_name`), if we search by giving exact values for all the three fields, search time is minimal—but if we provide the values for `city` and `first_name` only, the search uses only the `city` field to retrieve all matched records. Then a sequential lookup checks the matching with `first_name`. So, to improve the performance, one must ensure that the index is created on the order of search columns.

Applications and limitations [\[edit\]](#)

Indexes are useful for many applications but come with some limitations. Consider the following SQL statement: `SELECT first_name FROM people WHERE last_name = 'Smith'`. To process this statement without an index the database software must look at the `last_name` column on every row in the table (this is known as a [full table scan](#)). With an index the database simply follows the [B-tree](#) data structure until the Smith entry has been found; this is much less computationally expensive than a full table scan.

Consider this SQL statement: `SELECT email_address FROM customers WHERE email_address LIKE '%@wikipedia.org'`. This query would yield an email address for every customer whose email address ends with "@wikipedia.org", but even if the `email_address` column has been indexed the database must perform a full index scan. This is because the index is built with the assumption that words go from left to right. With a [wildcard](#) at the beginning of the search-term, the database software is unable to use the underlying B-tree data structure (in other words, the WHERE-clause is *not sargable*). This problem can be solved through the addition of another index created on `reverse(email_address)` and a SQL query like this: `SELECT email_address FROM customers WHERE reverse(email_address) LIKE reverse('%@wikipedia.org')`. This puts the wild-card at the right-most part of the query (now gro.aldepikw@%), which the index on `reverse(email_address)` can satisfy.

When the wildcard characters are used on both sides of the search word as `%wikipedia.org%`, the index available on this field is not used. Rather only a sequential search is performed, which takes O(N) time.

Types of indexes [\[edit\]](#)

Bitmap index [\[edit\]](#)

Main article: [Bitmap index](#)

A bitmap index is a special kind of indexing that stores the bulk of its data as [bit arrays](#) (bitmaps) and answers most queries by performing [bitwise logical operations](#) on these bitmaps. The most commonly used indexes, such as [B+ trees](#), are most efficient if the values they index do not repeat or repeat a small number of times. In contrast, the bitmap index is designed for cases where the values of a variable repeat very frequently. For example, the sex field in a customer database usually contains at most three distinct values: male, female or unknown (not recorded). For such variables, the bitmap index can have a significant performance advantage over the commonly used trees.

Dense index [\[edit\]](#)

A dense index in databases is a file with pairs of keys and [pointers](#) for every [record](#) in the data file. Every key in this file is associated with a particular pointer to a [record](#) in the sorted data file. In clustered indices with duplicate keys, the dense index points to the [first record](#) with that key.^[3]

Sparse index [\[edit\]](#)

A sparse index in databases is a file with pairs of keys and pointers for every [block](#) in the data file. Every key in this file is associated with a particular pointer to the [block](#) in the sorted data file. In clustered indices with duplicate keys, the sparse index points to the [lowest search key](#) in each block.

Reverse index [\[edit\]](#)

Main article: [Reverse index](#)

A reverse-key index reverses the key value before entering it in the index. E.g., the value 24538 becomes 83542 in the index. Reversing the key value is particularly useful for indexing data such as sequence numbers, where new key values monotonically increase.

Primary index [\[edit\]](#)

The primary index contains the key fields of the table and a pointer to the non-key fields of the table. The primary index is created automatically when the table is created in the database.

Secondary index [\[edit\]](#)

It is used to index fields that are neither ordering fields nor key fields (there is no assurance that the file is organized on key field or primary key field). One index entry for every tuple in the data file (dense index) contains the value of the indexed attribute and pointer to the block/record.

Index implementations [\[edit\]](#)

Indices can be implemented using a variety of data structures. Popular indices include [balanced trees](#), [B+ trees](#) and [hashes](#).^[4]

In [Microsoft SQL Server](#), the [leaf node](#) of the clustered index corresponds to the actual data, not simply a pointer to data that resides elsewhere, as is the case with a non-clustered index.^[5] Each relation can have a single clustered index and many unclustered indices.^[6]

Index concurrency control [\[edit\]](#)

Main article: [Index locking](#)

An index is typically being accessed concurrently by several transactions and processes, and thus needs [concurrency control](#). While in principle indexes can utilize the common database concurrency control methods, specialized concurrency control methods for indexes exist, which are applied in conjunction with the common methods for a substantial performance gain.

Covering index [\[edit\]](#)

In most cases, an index is used to quickly locate the data record(s) from which the required data is read. In other words, the index is only used to locate data records in the table and not to return data. A covering index is a special case where the index itself contains the required data field(s) and can answer the required data.

Consider the following table (other fields omitted):

ID	Name	Other Fields
12	Plug	...
13	Lamp	...
14	Fuse	...

To find the Name for ID 13, an index on (ID) is useful, but the record must still be read to get the Name. However, an index on (ID, Name) contains the required data field and eliminates the need to look up the record.

Covering indexes are each for a specific table. Queries which JOIN/ access across multiple tables, may potentially consider covering indexes on more than one of these tables.^[7]

A covering index can dramatically speed up data retrieval but may itself be large due to the additional keys, which slow down data insertion & update. To reduce such index size, some systems allow including non-key fields in the index. Non-key fields are not themselves part of the index ordering but only included at the leaf level, allowing for a covering index with less overall index size.

Standardization [edit]

No standard defines how to create indexes, because the ISO SQL Standard does not cover physical aspects. Indexes are one of the physical parts of database conception among others like storage (tablespace or filegroups). RDBMS vendors all give a CREATE INDEX syntax with some specific options that depend on their software's capabilities.

See also [edit]

- [Index locking](#)
- [Index \(search engine\)](#)
- [Inverted Index](#)

References [edit]

1. ^ [PostgreSQL 9.1.2 Documentation: CREATE TABLE](#)
2. ^ [Overview of Clusters](#) Oracle® Database Concepts 10g Release 1 (10.1)
3. ^ [Database Systems: The Complete Book](#). Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer D. Widom
4. ^ Gavin Powell (2006). *Chapter 8: Building Fast-Performing Database Models*. *Beginning Database Design*. Wrox Publishing. ISBN 978-0-7645-7490-0.
5. ^ "Clustered Index Structures". [SQL Server 2005 Books Online \(September 2007\)](#).
6. ^ Daren Bieniek, Randy Hess, Mike Hotek, Javier Loria, Adam Machanic, Antonio Soto, Adolfo Wiernik (January 2006). "Chapter 4: Creating Indices". *SQL Server 2005 Implementation and Management*. Microsoft Press.
7. ^ [Covering Indexes for Query Optimization](#)

Database management systems		[hide]
Types	Object-oriented (comparison) · Relational (comparison) · Document-oriented · Graph · NoSQL · NewSQL	
Concepts	Database · ACID · Armstrong's axioms · CAP theorem · CRUD · Null · Candidate key · Foreign key · Superkey · Surrogate key · Unique key	
Objects	Relation (table · column · row) · View · Transaction · Transaction log · Trigger · Index · Stored procedure · Cursor · Partition	
Components	Concurrency control · Data dictionary · JDBC · XQJ · ODBC · Query language · Query optimizer · Query rewriting system · Query plan	
Functions	Administration · Query optimization · Replication	
Related topics	Database models · Database normalization · Database storage · Distributed database · Federated database system · Referential integrity · Relational algebra · Relational calculus · Relational database · Relational model · Object-relational database · Transaction processing	

Categories: [Database management systems](#) | [Databases](#) | [Database index techniques](#)

This page was last edited on 15 November 2019, at 16:35 (UTC).

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Statistics](#) [Cookie statement](#) [Mobile view](#)

