Database Indexes

In a conventional RDBMS, indexes are special datasets that help to greatly improve performance of selected queries. Consider this illustration of a small table:

Row	ID	First Name	Last Name	Department
1	96	Ada	Brandt	Engineering
2	30	Zehra	Yavuz	Sales
3	98	Yves	Laurent	Sales
4	41	Hamza	Demir	Accounting
5	50	Sophia	Petrov	Sales
6	29	Cyril	Fedorov	Engineering
7	22	Akari	Tanaka	Accounting
8	83	Goro	Otsuka	Engineering

With such a small number of rows, you can find data from this table with ease. Suppose you want details for the person with ID 50, or you want to know how often the last name "Jones" appears in the table: you scan all the rows in the table to quickly answer your question. Now suppose that the table contains eight thousand rows: you could still answer your questions by scanning the rows, but it would take far too much time to view so much data.

A database system requires time to scan rows as well, and query time is especially affected by the time it takes to locate and read the needed data from disk storage. For a computer system, the time to read eight thousand rows may be no more than a few seconds, but the time becomes much worse

with eight million or eight billion rows. Database systems often allow analysts to improve query time on large tables by implementing database indexes.

Indexes and Query Performance

Look at this figure, which illustrates an index on the ID column of the small table above:

ID	Row
22	7
29	6
30	2
41	4
50	5
83	8
96	1
98	3

Notice that the ID column is ordered, and that each record in the index shows the row where that ID occurs. You can use the fact of ordering to quickly locate ID 50 in the index. This gives you row number 5, which you can then find quickly in the original table for details about the person with ID 50.

An index like this would allow you to quickly find the data for any person given their ID, even if the table contained eight thousand rows. For a larger table, it would take more time to locate the ID, and then additional time to find the correct row in the original table, but this two-step lookup

procedure—find the row number for ID 50 in the index, then locate the table row with that row number—would take far less time than a scan of the whole table without any index.

Now look at this illustration of an index on the Last Name column:

Last Name	Row
Brandt	1
Demir	4
Fedorov	6
Laurent	3
Otsuka	8
Petrov	5
Tanaka	7
Yavuz	2

With this index on Last Name, you can quickly determine that the last name Jones does not appear in the table at all. Without the index, you would need to scan all the rows of the table to determine how many times the last name Jones appeared. With the index, you only need to look for where Jones would appear, alphabetically, rather than looking at every row.

Sometimes an index can be even more compact:

Department	Row(s)
Accounting	4, 7
Engineering	1, 6, 8
Sales	2, 3, 5

With this index, you can quickly find the number of people in each department, or the rows with details about those people in the original table. It's worth noting that a suitable index built from your original table can allow you to answer some questions without consulting the original table at all. In this case, a count of persons in each department is available in the index itself.

Similar to the examples above, an index in a database system is a dataset, maintained along with the table being indexed, that can greatly improve the time it takes to find some data in the table. In order for the index to help accelerate table access, a few features are required:

1. **Random access** The system must have some capability of *random access* of rows in the table. That is, the system should be able to move to some spot partway in the table and read rows from there, without having to perform a disk read from the beginning of the table. In the initial table, you see a row number for each row. Knowing the row number, you can quickly locate and read the row of interest to you. Similarly, a database system will store some kind of *location information,* like a row number, or file block number, that the system can then use to quickly locate and read the row without scanning the entire table.
2. **Well ordered index data** In the table that indexes on ID, the ID values are ordered, and this allows you—or a computer system—to quickly locate any ID value in the list. Given any random row in the index, you can compare its ID value to the one you want and judge right away whether the entry you are seeking falls before or after your current position. This means that even with very large indexes, you can quickly locate the entry you are seeking with very few reads into the index.
3. **Alignment of index and table** Of course, if the index is to be of use to you, the records in the index must correspond exactly with the rows in the table being indexed. Otherwise, the index might cause you to attempt to read rows that are not present in the table, or worse, might lead you to miss rows that are present in the table! Questions addressed using an index that is not perfectly aligned with the original table are likely to yield incorrect answers.

4. **Indexing on the right fields** In the examples given here, the index on ID aids access to the data *when you reference records by ID*. The index on last name helps with access when referencing records by that column. However, if you ask a question about first name, there is no indexing support to improve the performance of a query on that column, unless you build another index for that purpose. Every index you define requires additional space and compute time, so it is important for you to judge which columns would warrant indexing—those you will use frequently, or in queries that you need to run quickly.

Indexes and Key Constraints

In addition to using indexes to aid query performance, many systems that have table indexing will use unique indexes as a way to enforce primary key constraints. A unique index will permit no more than one entry for its key value (the value being indexed). Because the indexed items are well organized, the system can very quickly determine whether any given value is present in the index. By creating and maintaining such an index on the primary key column of a table, the system can quickly determine if some new row submitted for insert repeats an existing primary key, and it can reject that row right away. It is fortuitous that the primary key is also a common value to use for finding a row in a table, so the unique index on primary key does two things: it helps to enforce uniqueness on the primary key values, and it delivers speed to many queries.

Foreign keys are also often enforced with the help of indexes. A database system will often create an index on a column when that column is declared as a foreign key. Consider, for example, a **department** table, and a related **employee** table. Each record in the **employee** table has a **department_id** as a foreign key to the **department** table. If the system builds an index on **employee.department_id**, then it becomes a simple matter to enforce some foreign key constraints: if a user attempts to delete a **department** row, a quick check of the index on **employee.department_id** can determine whether this delete would leave some employees without a department, and so whether the delete should be prevented.

Foreign key constraints may work in a different way from the one above. In some pairs of related tables, such as an **order** table and a **line_item** table, the foreign key of **line_item.order_id** can be used to efficiently support a *cascading delete*. That is, whenever an order is deleted, the line items for that order have no reason for being; so the system can use the **id** of that order and the **line_item.order_id** index to quickly find the companion rows in the **line_item** table and delete them.

Maintaining Indexes

Notice that every index on a table requires a dataset in addition to the original table, and that the index necessarily contains data that is a *repeat* of the table data. To make use of the index, there must always be a way to maintain consistency between the index and the indexed table—point 3, "alignment of index and table," in the numbered list above. Some systems maintain a table's indexes automatically, along with changes to the table, and some do not.

In the case of key constraints, it becomes critical that the maintenance of indexes occurs automatically and *transactionally,* at the same time as DML on the indexed tables. For example, a unique index must be checked and updated at the time of each row insert, in order to correctly aid in the enforcement of a unique constraint on primary key columns.

In addition to supporting key constraints, it can be helpful if the system provides a guarantee that indexes are always automatically synchronized with their tables. Operational databases often provide automatic synchronization of tables and indexes, and they rely on database transactional capabilities to do so.

In some systems, indexes are not updated automatically at the same time as table DML. In these systems, an index will become "stale" and inconsistent with its table whenever any DML on the table is performed. In these systems, you will perform a rebuild of all needed indexes as part of ETL: load tables with new or updated data, then rebuild indexes prior to any query activity.

To summarize, there are two ways that indexes may be maintained:

- **Transactionally, in lock-step with table DML:** This will impose some performance cost on every DML statement. In exchange, the system can support primary key and foreign key constraints and always help to accelerate queries. This is good for operational databases.
- **In deferred rebuild, separate from table DML:** This does not help support key constraints at all. To use indexes for runtime performance, you must rebuild the indexes as part of data preparation before running your queries. This approach occurs often with analytic database systems.

A Final Note on Indexes with Big Data

It is worth noting that the Apache Hive project has had only limited support for indexes on tables in the big data space. In Hive, indexes are supported only on specific file formats, and only with deferred rebuilds. A primary obstacle is the inability of many file systems to provide random access to data, point 1 in the numbered list above. In fact, as of Apache Hive version 3.0, the project abandoned support for indexes altogether. (Many companies are still using older versions.) Analytic systems in big data accelerate query performance with other techniques, such as extremely efficient columnar file formats, like Apache Parquet, and optimized query engines like Apache Impala. You'll learn more about these later.

When enterprises need extreme speed on queries, they may choose to invest in complete, dedicated indexing systems, using software such as Solr or Elasticsearch, both of which use Apache Lucene indexing internally.