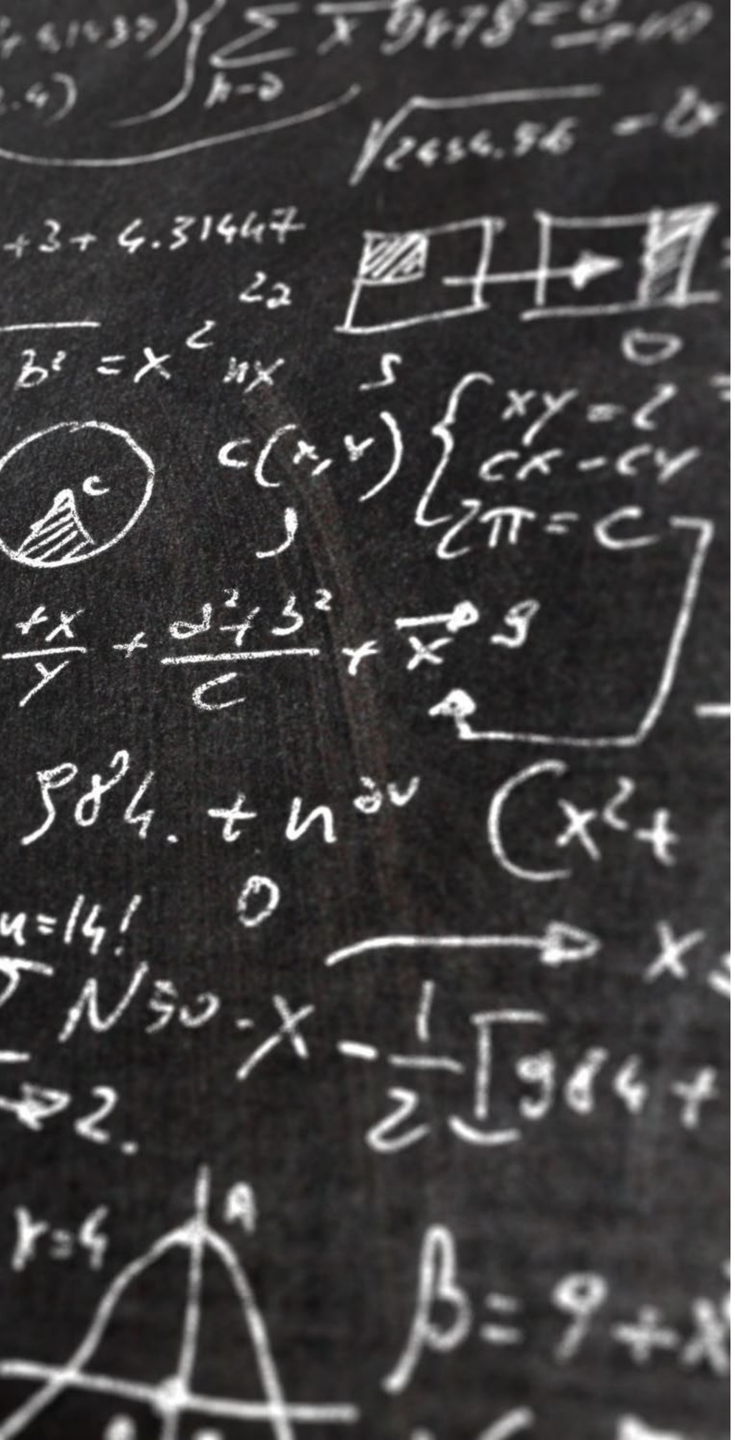
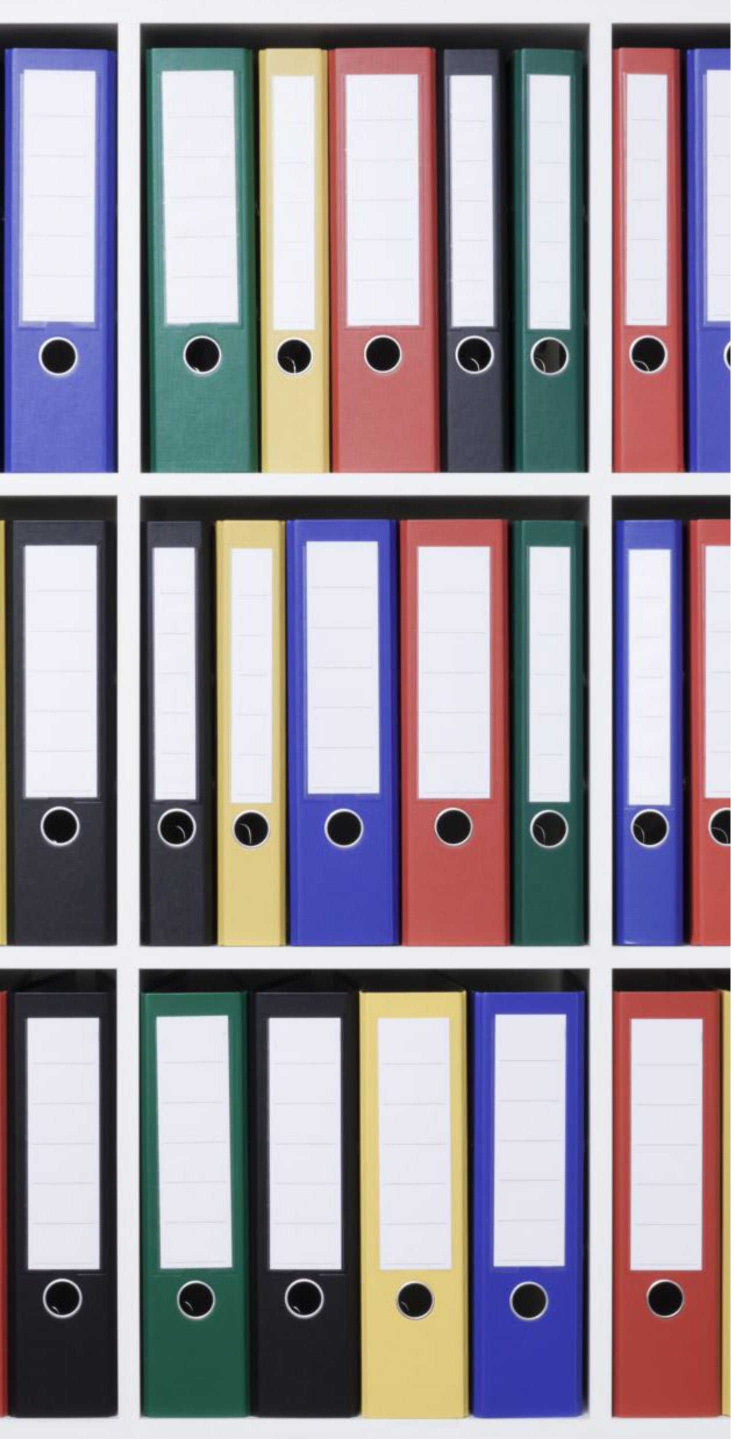
Improving Database Performance Indexes in Action

Advanced Databases



Index in Action

- Indexes are schema objects that are logically and physically separate from the data in objects with which they are associated
  - Therefore we can delete an index without impacting the stored data



Index in Action

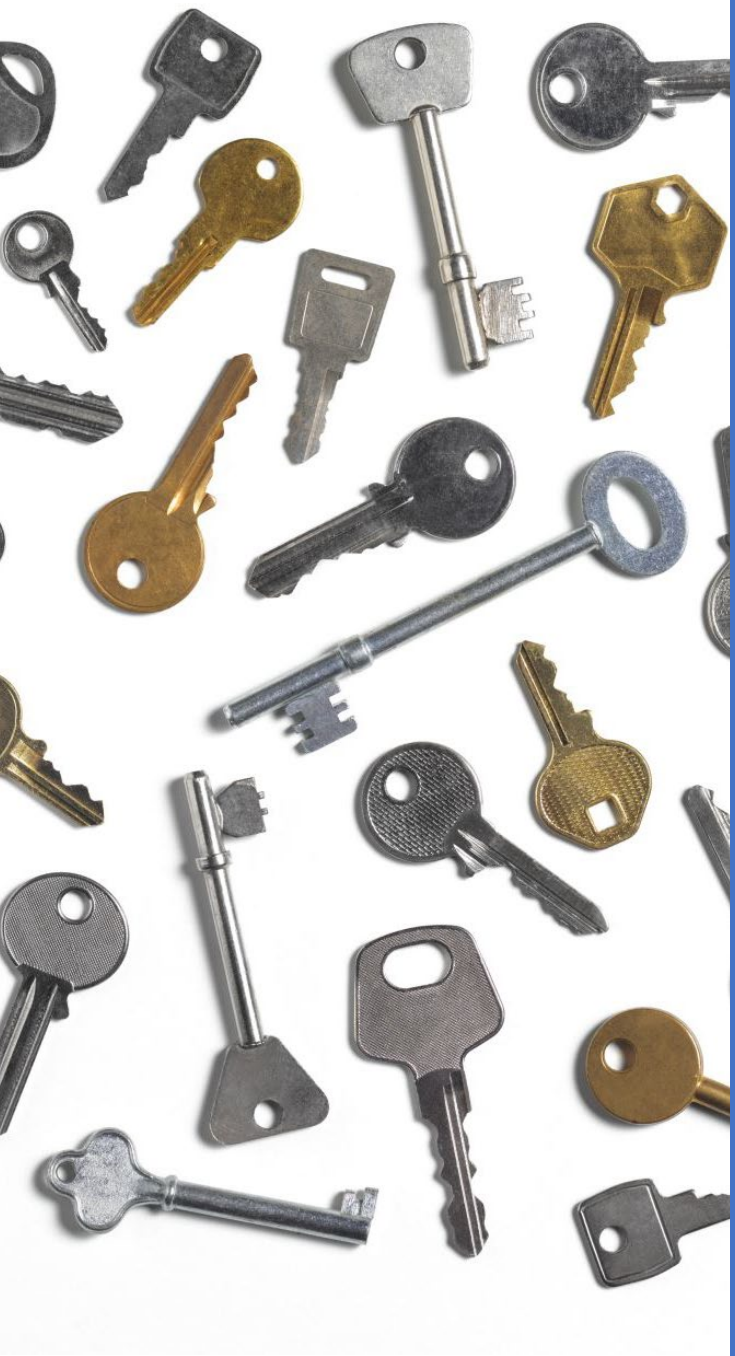
- Indexes do not affect how you write your SQL statements
- An index is a fast access path to a single row of data.
- It affects only the query plan, if an index exists then it is available for use in a query plan:
  - Given a data value that has been indexed, the index points directly to the location of the rows containing that value.

Index in Action

- The database automatically maintains and uses indexes after they are created.
- The database also automatically reflects changes to data, such as adding, updating, and deleting rows, in all relevant indexes with no additional actions required by users.
- Retrieval performance of indexed data remains almost constant, even as rows are inserted.
- However, the presence of many indexes on a table degrades DML performance because the database must also update the indexes.

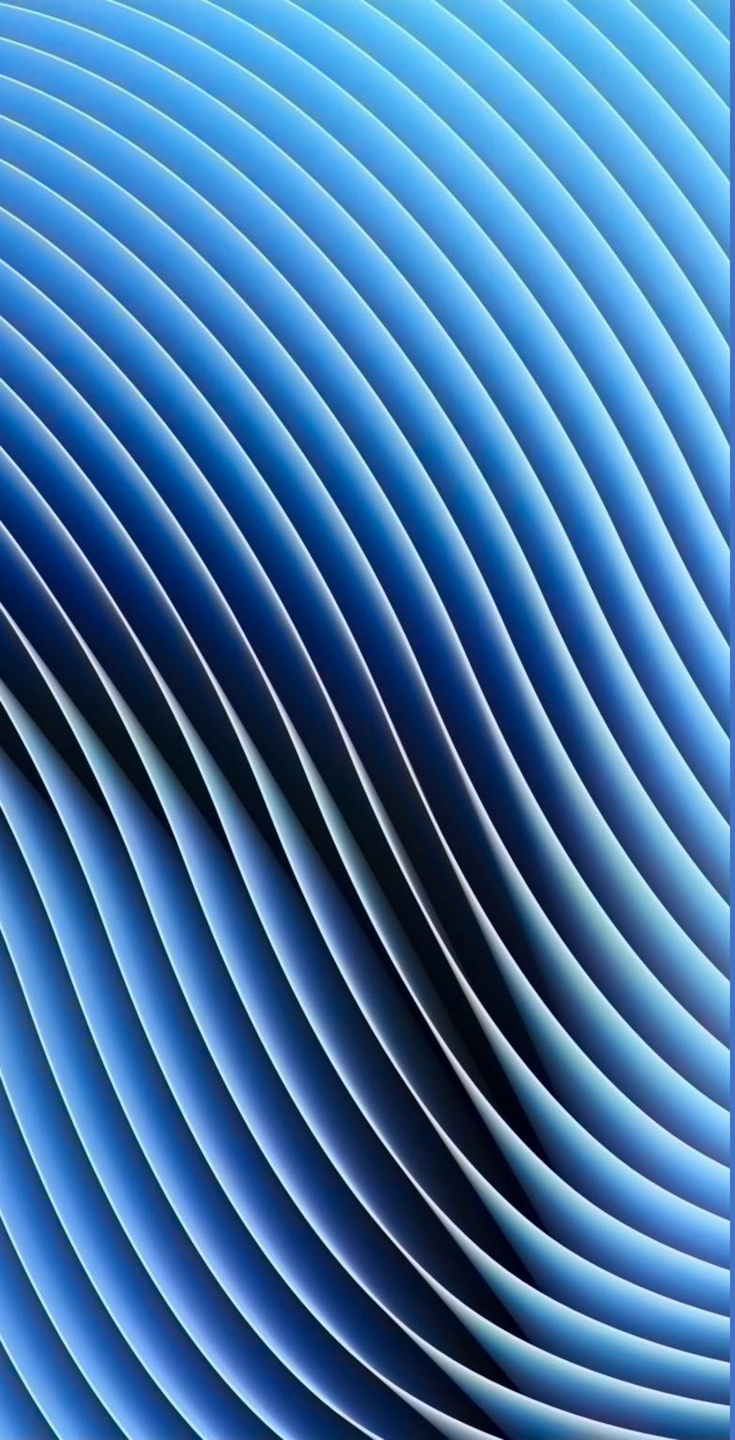
Index in Action

- **Unique index**
- Makes sure that your table does not have more than one row with the same value.
- Helpful when it comes to maintaining data integrity and high performance in PostgreSQL.



Index in Action

- **Note**: Primary and unique keys automatically have unique indexes, but you might want to create a unique index on a foreign key or some non-key attributes.

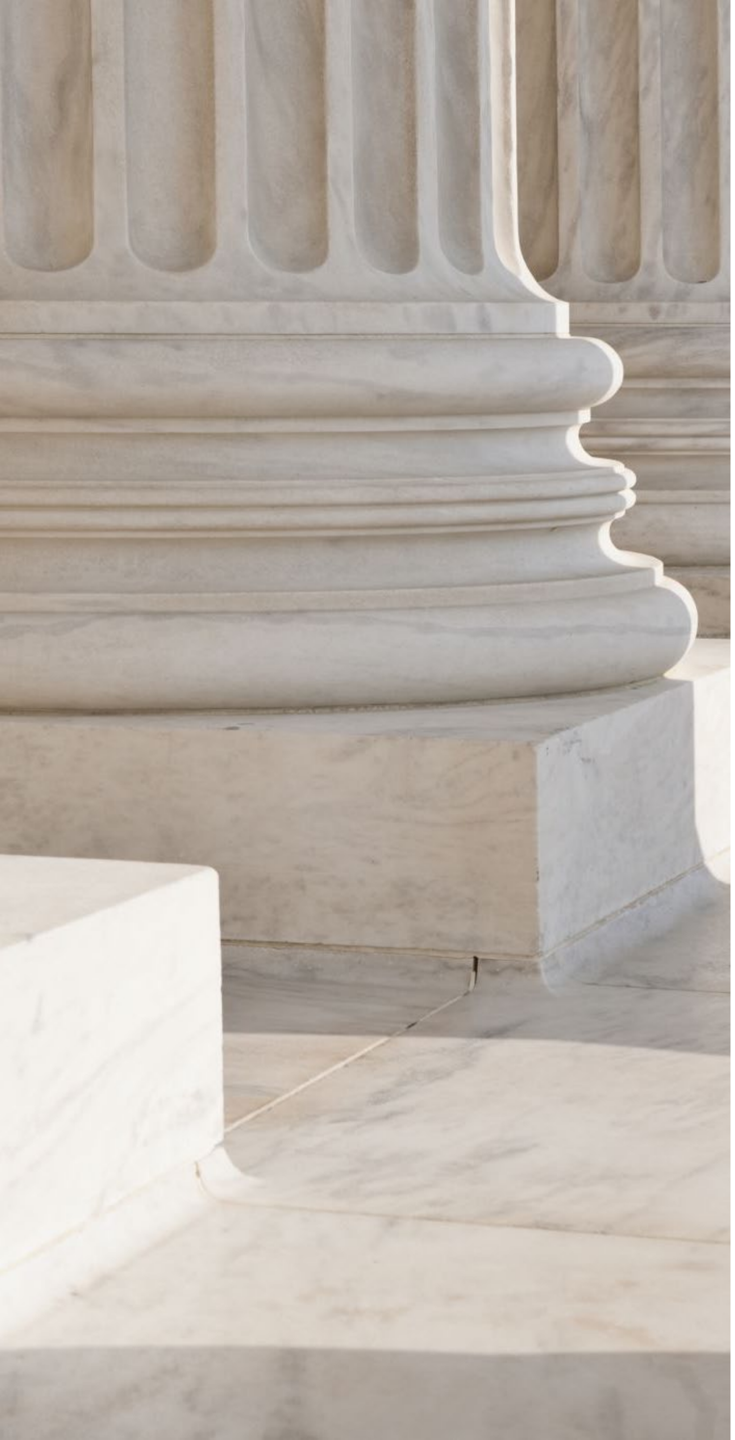


Index in Action

CREATE UNIQUE INDEX vip_customer_ix ON vip_customers (customer_id);

customer_id is a foreign key in the vip_customers table

However, this will only allow one occurrence of each value of customer_id in vip_customers



Index in Action

- **Multi-column index**
- An index can be created on more than one table column.
    - Limited to 32 columns generally
    - Only B-Tree, GIN, GiST, and BRIN types support multi-column indexes.
- Columns in the index should appear in the order that makes the most sense for the queries that will retrieve data and need not be adjacent in the table.
- Composite indexes can speed retrieval of data for SELECT statements in which the WHERE (predicate) clause references all or the leading portion of the columns in the composite index.
- Therefore, the order of the columns used in the definition is important.
    - In general, the most commonly accessed columns go first.

Index in Action

- For example, suppose an application frequently queries the last_name, job_id, and salary columns in the employees table.
- Also assume that last_name has high cardinality, which means that the number of distinct values is large compared to the number of table rows.
- You create an index with the following column order:

```
CREATE INDEX employees_ix
   ON employees (last_name,
job_id, salary);
```

- Queries that access the following could use this index:
  - all three columns,
  - only the last_name column,
  - or only the last_name and
  - job_id columns

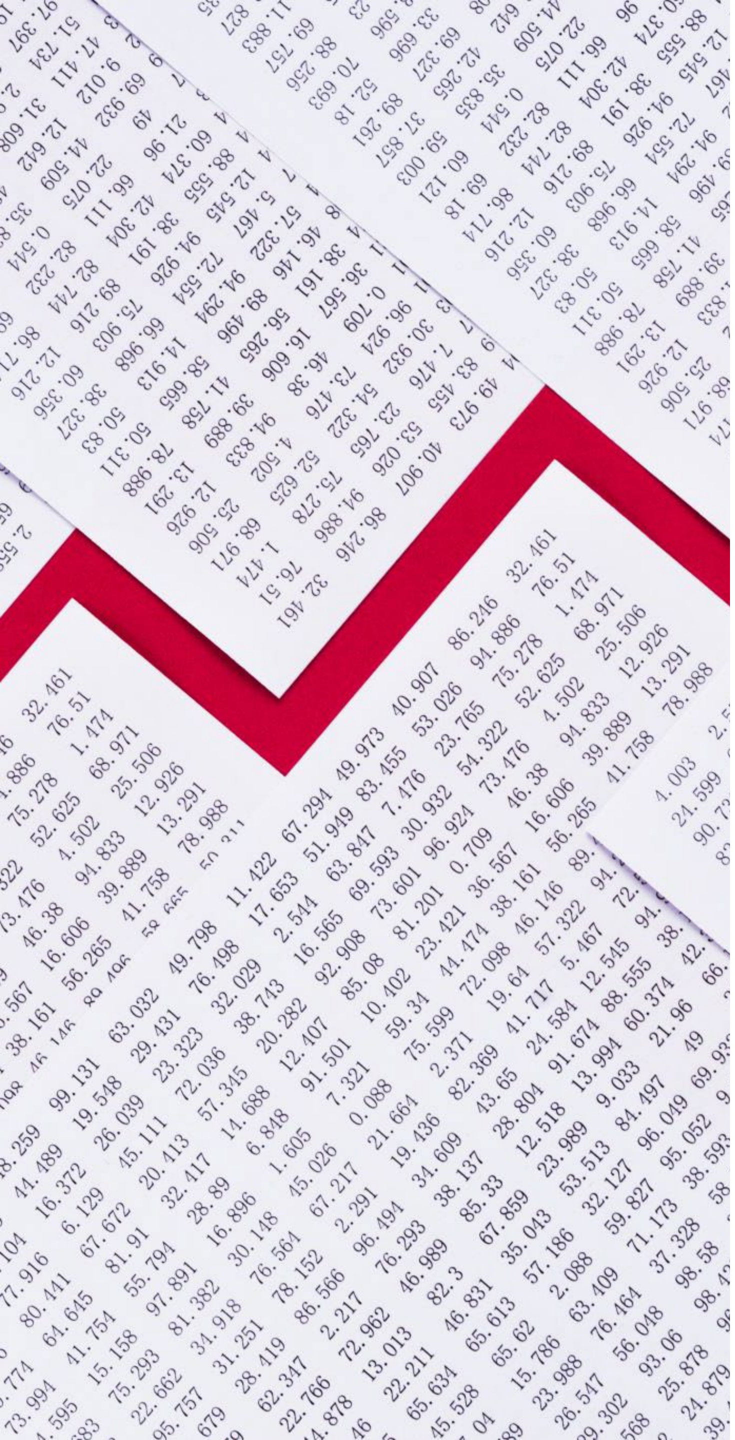
Index in Action

Partial index

An index with a WHERE clause, which covers a particular subset of data in a table. It is rather small, works faster, and can be used alongside other indexes on more complex queries.

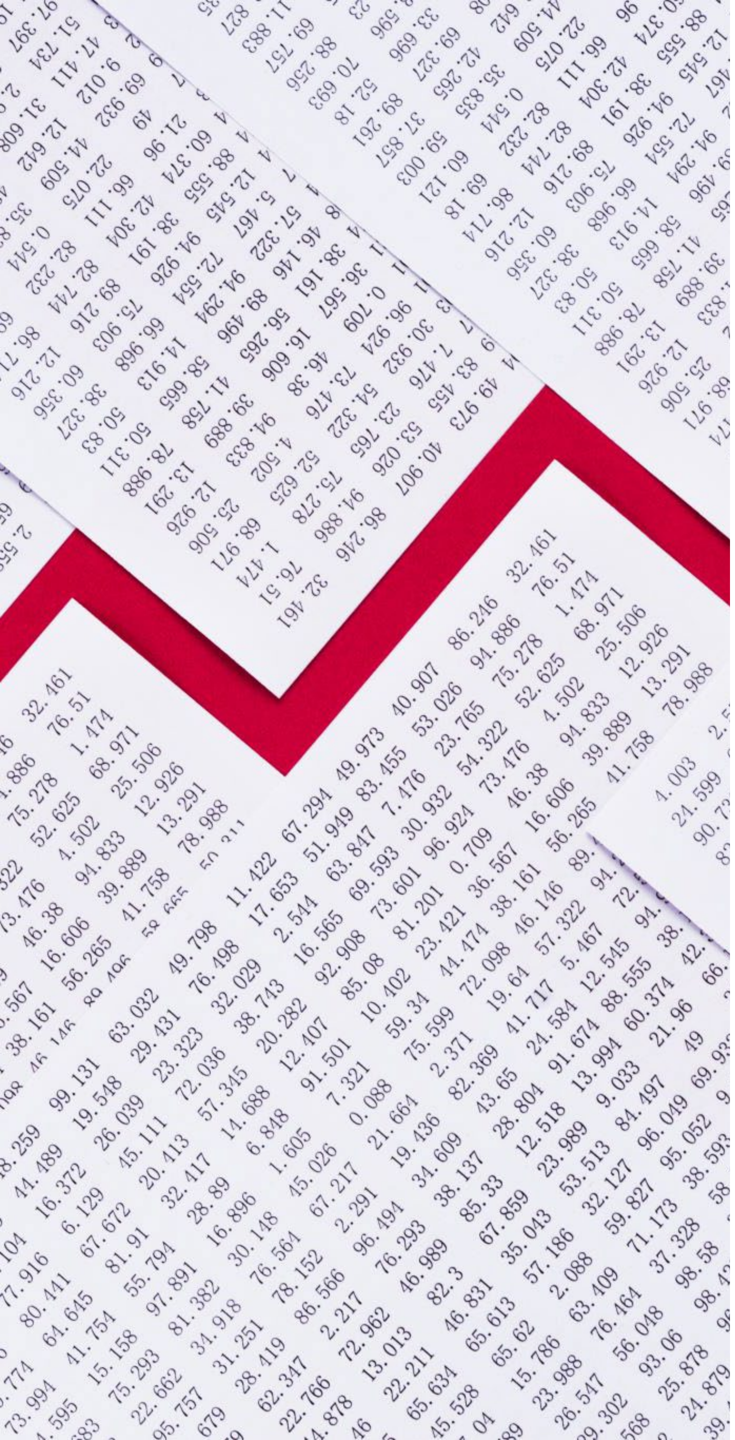
Index in Action

- Adding a partial index
- `CREATE INDEX index_name ON table_name(column_list) WHERE condition;`
- `create index vip_cust_idx on customer (VIP) where VIP=‘N’;`



Index in Action

- **Expression index**
- Useful for queries matching on a function or modification of data.
- PostgreSQL makes it possible to index the result of the function and make this search as efficient as search by raw data values.



Index in Action

- Adding an index expression
- `CREATE INDEX index_name ON table_name (expression);`
- `CREATE INDEX large_sales_idx on sales(amount) where amount > 400;`

Index in Action

- Multiple indexes can exist for the same table if the permutation of columns differs for each index.
- You can create multiple indexes using the same columns if you specify distinctly different permutations of the columns.
- For example, the following SQL statements are both valid:

```
CREATE INDEX employee_idx1
ON employees (last_name,
job_id);
```

```
CREATE INDEX employee_idx2
ON employees (job_id,
last_name);
```

Index in Action

- You should always properly analyze your workload using query execution plans to determine the suitable Postgres index type.
- Always create indexes on the most executed and costly queries.
  - Avoid creating an index to satisfy a specific query.
- As per best practice, always define a primary or unique key in a Postgres table.
  - It automatically creates the B-tree index.
- Avoid creating multiple indexes on a single column.
  - It is better to look at which index is appropriate for your workload and drop the unnecessary indexes.
- There is no specific limit on the number of indexes in the table; however, try to create the minimum indexes satisfying your workload.

Index in Action

- **B-Tree**
- default index in Postgres
- best used for specific value searches, scanning ranges, data sorting or pattern matching
- **Hash**
- best suited to work with equality operators.
- equality operator looks for the exact match of data
- **GiST**
- can index complex data such as geometric data and network address data
- **BRIN**
- stores summary of blocks (minimum value, maximum value and page number)
- useful for extensive data such as timestamps and temperature sensor data
- **GIN**
- for data types whose values are not atomic, but consist of elements

Why is my query not using an index?

- There are many reasons why the Postgres planner may choose to not use an index.
- Most of the time, the planner chooses correctly, even if it isn’t obvious why.
- It’s okay if the same query uses an index scan on some occasions but not others.
- The number of rows retrieved from the table may vary based on the particular constant values the query retrieves.
- So, for example, it might be correct for the query planner to use an index for the query select * from foo where bar = 1, and yet not use one for the query select * from foo where bar = 2, if there happened to be far more rows with “bar” values of 2.
- When this happens, a sequential scan is actually most likely much faster than an index scan, so the query planner has in fact correctly judged that the cost of performing the query that way is lower.