Indexes

Overview

- Introduction
- Index types
- Multicolumn indexes
- Indexes and ORDER BY
- Unique indexes
- Indexes on Expressions

Index

- Indexes are a common way to enhance database performance.

- An index allows the database server to find and retrieve specific rows much faster than it could do without an index.

- But indexes also add overhead to the database system as a whole, so they should be used sensibly.

Index

- Suppose we have a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

- and the application issues many queries of the form:

```
SELECT content FROM test1 WHERE id = constant;
```

Index

- With no advance preparation, the system would have to scan the entire `test1` table, row by row, to find all matching entries.

- If there are many rows in `test1` and only a few rows (perhaps zero or one) that would be returned by such a query, this is clearly an inefficient method.

- But if the system has been instructed to maintain an index on the `id` column, it can use a more efficient method for locating matching rows.

Index

- A similar approach is used in most non-fiction books: terms and concepts that are frequently looked up by readers are collected in an alphabetic index at the end of the book.

- The interested reader can scan the index relatively quickly and flip to the appropriate page(s), rather than having to read the entire book to find the material of interest.

- Just as it is the task of the author to anticipate the items that readers are likely to look up, it is the task of the database programmer to foresee which indexes will be useful.

Index

- The following command can be used to create an index on the id column, as discussed:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Index

- The name test1_id_index can be chosen freely, but you should pick something that enables you to remember later what the index was for.
- To remove an index, use the DROP INDEX command.

```
DROP INDEX index_name;
```

- Indexes can be added to and removed from tables at any time.

Index

- Once an index is created, no further intervention is required
- The system will update the index when the table is modified
- And it will use the index in queries when it thinks doing so would be more efficient than a sequential table scan.
- But you might have to run the ANALYZE command regularly to update statistics to allow the query planner to make educated decisions.

Index

- Indexes can also benefit UPDATE and DELETE commands with search conditions.

- Indexes can moreover be used in join searches.

- Thus, an index defined on a column that is part of a join condition can also significantly speed up queries with joins.

Index

- After an index is created, the system has to keep it synchronized with the table.
- This adds overhead to data manipulation operations.
- Therefore indexes that are seldom or never used in queries should be removed.

How index works:

Index column

9	Alphonso
2	Arvy
7	Byrle
4	Con
10	Fairfax
1	Hilde
8	Howard
3	Reinald
6	Tildy
11	Vanda
5	Wayne

Table column

	passenger_id [PK] integer	first_name character varying (50)
1	1	Hilde
2	2	Arvy
3	3	Reinald
4	4	Con
5	5	Wayne
6	6	Tildy
7	7	Byrle
8	8	Howard
9	9	Alphonso
10	10	Fairfax
11	11	Vanda

Index types

- B-tree
- Hash
- GiST
- SP-GiST
- GIN
- BRIN

Index types

- Each index type uses a different algorithm that is best suited to different types of queries.

- By default, the `CREATE INDEX` command creates B-tree indexes, which fit the most common situations.

B-tree index

- B-trees can handle equality and range queries on data that can be sorted into some ordering.

- PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:

- < <= = >= >

B-tree index

- B-tree indexes can also be used to retrieve data in sorted order.

- This is not always faster than a simple scan and sort, but it is often helpful.

Hash index

- Hash indexes can only handle simple equality comparisons.

- The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the = operator. The following command is used to create a hash index:

```
CREATE INDEX name ON table USING HASH (column);
```

GiST index

- GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented.
- Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the operator class).

```
<<   &<   &>   >>   <<|   &<|   |&>   |>>   @>   <@   ~=   &&
```

Multicolumn Indexes

- An index can be defined on more than one column of a table. For example, if you have a table of this form:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

- and you frequently issue queries like:

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

Multicolumn Indexes

- then it might be appropriate to define an index on the columns major and minor together, e.g.:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

- Up to 32 columns can be specified. (This limit can be altered when building PostgreSQL; see the file `pg_config_manual.h`.)

Indexes and ORDER BY

- In addition to simply finding the rows to be returned by a query, an index may be able to deliver them in a specific sorted order.

- only B-tree can produce sorted output

- By default, B-tree indexes store their entries in ascending order with nulls last.

Indexes and ORDER BY

- You can adjust the ordering of a B-tree index by including the options `ASC`, `DESC`, `NULLS FIRST`, and/or `NULLS LAST` when creating the index; for example:

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);

CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

Unique Indexes

- Indexes can also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

- Currently, only B-tree indexes can be declared unique.

Unique Indexes

- When an index is declared unique, multiple table rows with equal indexed values are not allowed.
- Null values are not considered equal.
- A multicolumn unique index will only reject cases where all indexed columns are equal in multiple rows.

Unique Indexes

- PostgreSQL automatically creates a unique index when a unique constraint or primary key is defined for a table.

- The index covers the columns that make up the primary key or unique constraint (a multicolumn index, if appropriate), and is the mechanism that enforces the constraint.

Indexes on Expressions

- An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table.

- This feature is useful to obtain fast access to tables based on the results of computations.

Indexes on Expressions

- For example, a common way to do case-insensitive comparisons is to use the lower function:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

- This query can use an index if one has been defined on the result of the lower(col1) function:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Indexes on Expressions

```sql
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

```sql
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

Index Type	Use Cases	Supported Operations	Strengths	Limitations
B-tree	General indexing, equality and range queries	`=`, `<`, `>`, `<=`, `>=`, `BETWEEN`	Fast, widely used, supports ordering	Less efficient on complex, multi-value data
Hash	Exact matches	`=`	Efficient for equality	No range queries, limited operator support
GIN	Multi-value types, arrays, JSONB, full-text search	`@>`, `?`, `?`	`	Best for multi-value data, JSONB, text search
GiST	Geometric/spatial data, range types	Custom operators like `<->`	Nearest neighbor, spatial, range searches	More complex, slower on simple equality
SP-GiST	Partitioned, hierarchical data (e.g., IPs)	Custom operators for partitioned data	Efficient for tree-like structures	Limited use cases
BRIN	Large, append-only tables, ordered data	Range queries over blocks	Space-efficient, fast for large tables	Not as precise, less accurate for specific queries

Examples

B-tree index for **flight** search

```
CREATE INDEX idx_flight_no ON flights (flight_no);
```

After indexing, queries will run faster because they won't require a full table scan.

```
SELECT * FROM flights WHERE flight_no = 'US-NY'
```

GIN index

```
ALTER TABLE passengers ADD COLUMN category jsonb;

CREATE INDEX idx_passenger_category ON passengers USING gin (category);
```

```
SELECT * FROM passengers WHERE category @> '{"category type": true}';
```

Hash index

```
CREATE INDEX idx_passport_number ON passengers USING hash (passport_number);
```

```
SELECT * FROM passengers WHERE passport_number = '11562593487'
```

BRIN index

```sql
CREATE INDEX idx_created_at ON booking USING brin (created_at);

SELECT * FROM booking
WHERE created_at BETWEEN '2024-01-01' AND '2024-01-31'
```

```sql
EXPLAIN SELECT * FROM booking
WHERE created_at BETWEEN '2024-01-01' AND '2024-01-31'
```

	QUERY PLAN text
1	Seq Scan on booking (cost=0.00..12.50 rows=57 width=46)
2	[...] Filter: ((created_at >= '2024-01-01'::date) AND (created_at <= '2024-01-31'::date))

GiST index

```
ALTER TABLE airport ADD COLUMN geolocation POINT;

CREATE INDEX idx_airport_location ON airport USING gist (geolocation);
```

```
SELECT * FROM airport
WHERE geolocation <@ CIRCLE '(37.618423, 55.755833), 10';
```

Show all indexes of a database

```sql
SELECT
    indexname AS index_name,
    indexdef AS definition
FROM
    pg_indexes
WHERE
    schemaname = 'public';
```

	index_name name	definition text
1	passengers_pkey	CREATE UNIQUE INDEX passengers_pkey ON public.passengers USING btree (passenger_id)
2	airport_pkey	CREATE UNIQUE INDEX airport_pkey ON public.airport USING btree (airport_id)
3	boarding_pass_pkey	CREATE UNIQUE INDEX boarding_pass_pkey ON public.boarding_pass USING btree (boarding_pass_id)
4	booking_pkey	CREATE UNIQUE INDEX booking_pkey ON public.booking USING btree (booking_id)
5	booking_flight_pkey	CREATE UNIQUE INDEX booking_flight_pkey ON public.booking_flight USING btree (booking_flight_id)
6	flights_pkey	CREATE UNIQUE INDEX flights_pkey ON public.flights USING btree (flight_id)
7	airline_pkey	CREATE UNIQUE INDEX airline_pkey ON public.airline USING btree (airline_id)
8	baggage_pkey	CREATE UNIQUE INDEX baggage_pkey ON public.baggage USING btree (baggage_id)
9	baggage_check_pkey	CREATE UNIQUE INDEX baggage_check_pkey ON public.baggage_check USING btree (baggage_check_id)
10	security_check_pkey	CREATE UNIQUE INDEX security_check_pkey ON public.security_check USING btree (security_check_id)
11	idx_flight_no	CREATE INDEX idx_flight_no ON public.flights USING btree (flight_no)
12	idx_passenger_category	CREATE INDEX idx_passenger_category ON public.passengers USING gin (category)
13	idx_passport_number	CREATE INDEX idx_passport_number ON public.passengers USING hash (passport_number)
14	idx_created_at	CREATE INDEX idx_created_at ON public.booking USING brin (created_at)

Show all indexes of a table

```
SELECT
    indexname AS index_name,
    indexdef AS definition
FROM
    pg_indexes
WHERE
    tablename = 'passengers';
```

	index_name name	definition text
1	passengers_pkey	CREATE UNIQUE INDEX passengers_pkey ON public.passengers USING btree (passenger_id)
2	idx_passenger_category	CREATE INDEX idx_passenger_category ON public.passengers USING gin (category)
3	idx_passport_number	CREATE INDEX idx_passport_number ON public.passengers USING hash (passport_number)

Monitoring index usage

```
SELECT * FROM pg_stat_user_indexes WHERE schemaname = 'public';
```

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	42032	42038	public	passengers	passengers_pkey	0	0	0
2	42011	42040	public	airport	airport_pkey	0	0	0
3	42020	42042	public	boarding_pass	boarding_pass_pkey	0	0	0
4	42026	42044	public	booking	booking_pkey	0	0	0
5	42023	42046	public	booking_flight	booking_flight_pkey	0	0	0
6	42029	42048	public	flights	flights_pkey	2	2	2
7	42008	42050	public	airline	airline_pkey	0	0	0
8	42017	42052	public	baggage	baggage_pkey	0	0	0
9	42014	42054	public	baggage_check	baggage_check_pkey	0	0	0
10	42035	42056	public	security_check	security_check_pkey	0	0	0
11	42029	42113	public	flights	idx_flight_no	2	3	0

Summary

Advantages	Disadvantages
Improved Query Performance	**Increased Storage Requirements**
- Faster Lookups	- Additional Disk Space
- Efficient Filtering	
Faster Join Operations	**Slower Write Operations**
- Optimized Joins	- Overhead for DML Operations
Enhanced Sorting and Grouping	**Maintenance Overhead**
- Quicker Sorting	- Regular Maintenance Needed
- Efficient Aggregations	