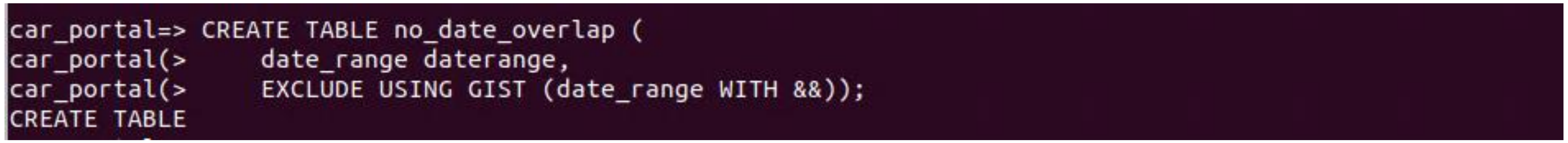
Indexes

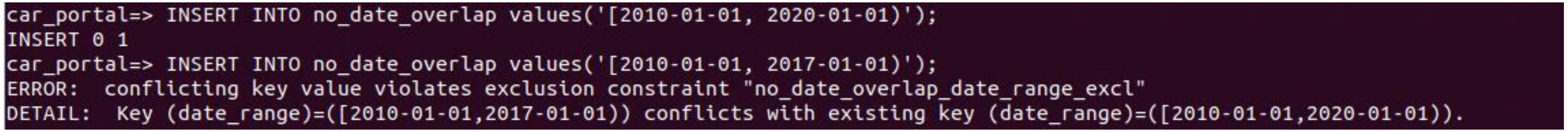
- An index is a physical database object that's defined in a table column or a list of columns.
- In PostgreSQL, there are many types of indexes and several ways to use them.
- Indexes can be used, in general, to do the following:
  - **Optimize performance**: An index allows for the efficient retrieval of a small number of rows from the table. Whether the number of rows is considered small is determined by the total number of rows in the table and the execution planner settings.
  - **Validate constraints**: An index can be used to validate the constraints on several rows. For example, the `UNIQUE` check constraint creates a unique index on the column behind the scenes.

- The following example shows how to use `GIST` to forbid overlapping between date ranges.
- Checking for date overlapping is very important in reservation systems, such as car and hotel reservation systems.
- For more information, take a look at https://www.postgresql.org/docs/current/static/rangetypes.html



```
car_portal=> CREATE TABLE no_date_overlap (
car_portal(>     date_range daterange,
car_portal(>     EXCLUDE USING GIST (date_range WITH &&));
CREATE TABLE
```

- To test for date range overlapping, let's insert some data, noting the syntax of the date range.
- The parentheses (or brackets) indicate whether the lower and upper bounds are exclusive or inclusive:



```
car_portal=> INSERT INTO no_date_overlap values('[2010-01-01, 2020-01-01)');
INSERT 0 1
car_portal=> INSERT INTO no_date_overlap values('[2010-01-01, 2017-01-01)');
ERROR:  conflicting key value violates exclusion constraint "no_date_overlap_date_range_excl"
DETAIL:  Key (date_range)=([2010-01-01,2017-01-01)) conflicts with existing key (date_range)=([2010-01-01,2020-01-01)).
```

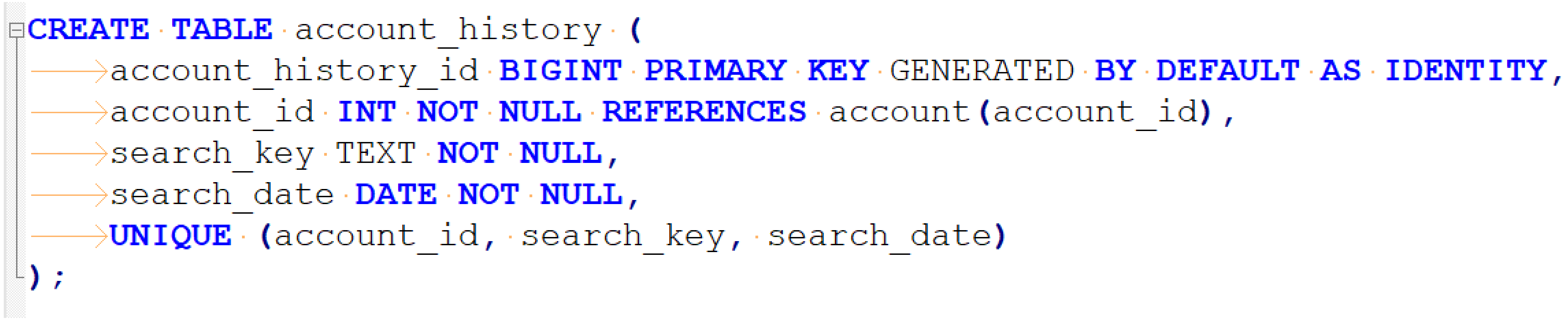
Index Synopses

- Indexes can be created using the `CREATE INDEX` statement.
- Since an index is a physical database object, you can specify the `TABLESPACE` and `storage_parameter` options.
- An index can be created on columns or expressions.
- Index entries can be sorted in an `ASC` or `DESC` order.
- Also, you can specify the sort order for `NULL` values.
- If an index is created for text fields, you can also specify the collation.
- The `ONLY` option, as shown in the following synopsis, is used to handle table inheritance and partitioning.
    - If a table is partitioned, the index will also be created in partitions, by default, when the `ONLY` option is not specified.

- The following is the synopsis for the index:
-
```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON [ ONLY ] table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

Index Selectivity

- Let's take a look at the `account_history` table in the car web portal example, as follows:



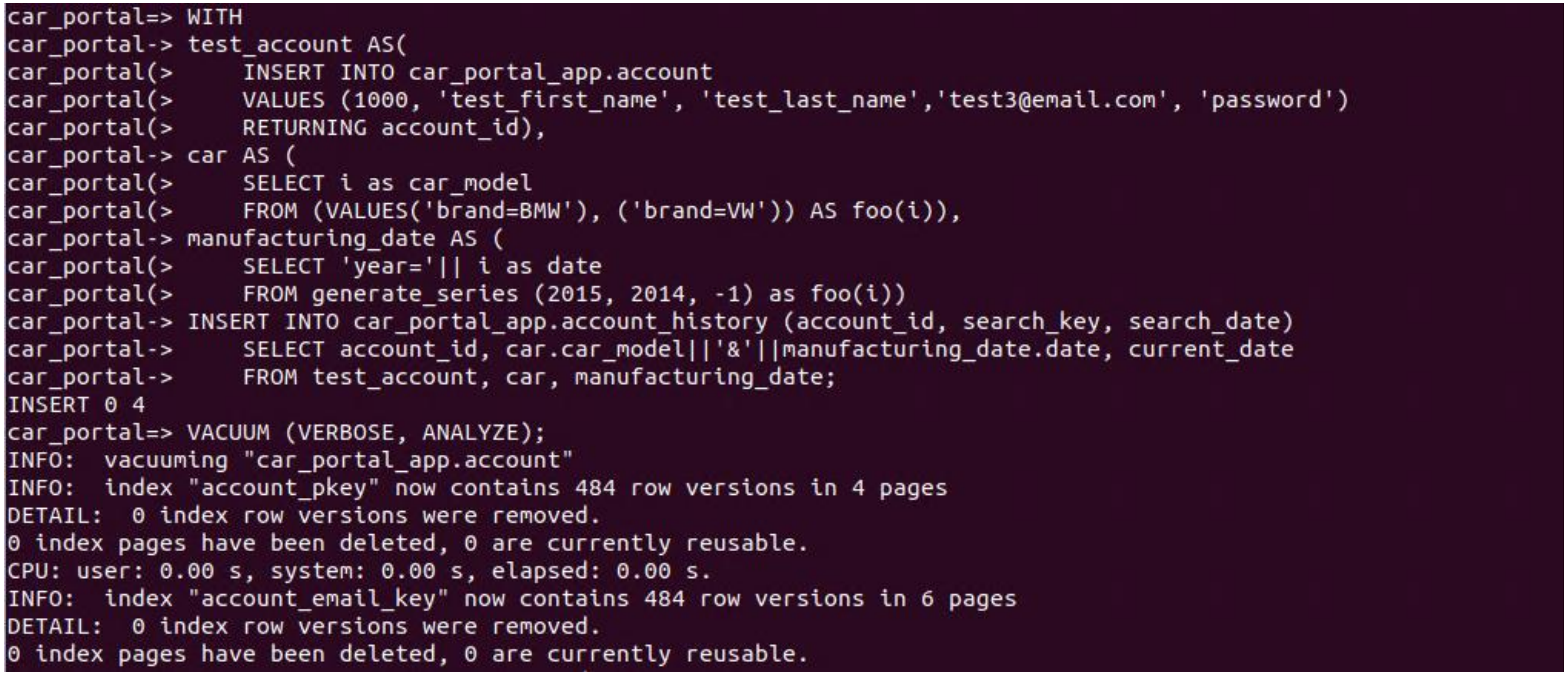
```sql
CREATE TABLE account_history (
	account_history_id BIGINT PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
	account_id INT NOT NULL REFERENCES account(account_id),
	search_key TEXT NOT NULL,
	search_date DATE NOT NULL,
	UNIQUE (account_id, search_key, search_date)
);
```

- The unique constraint, `UNIQUE (account_id, search_key, search_date)`, has two purposes.
  - The first purpose is to define the validity constraint for inserting the search key into the table only once each day, even if the user searches for the key several times.
  - The second purpose is to retrieve data quickly.

- Let's assume that we would like to show the last 10 searches for a user. The query for performing this would be as follows:

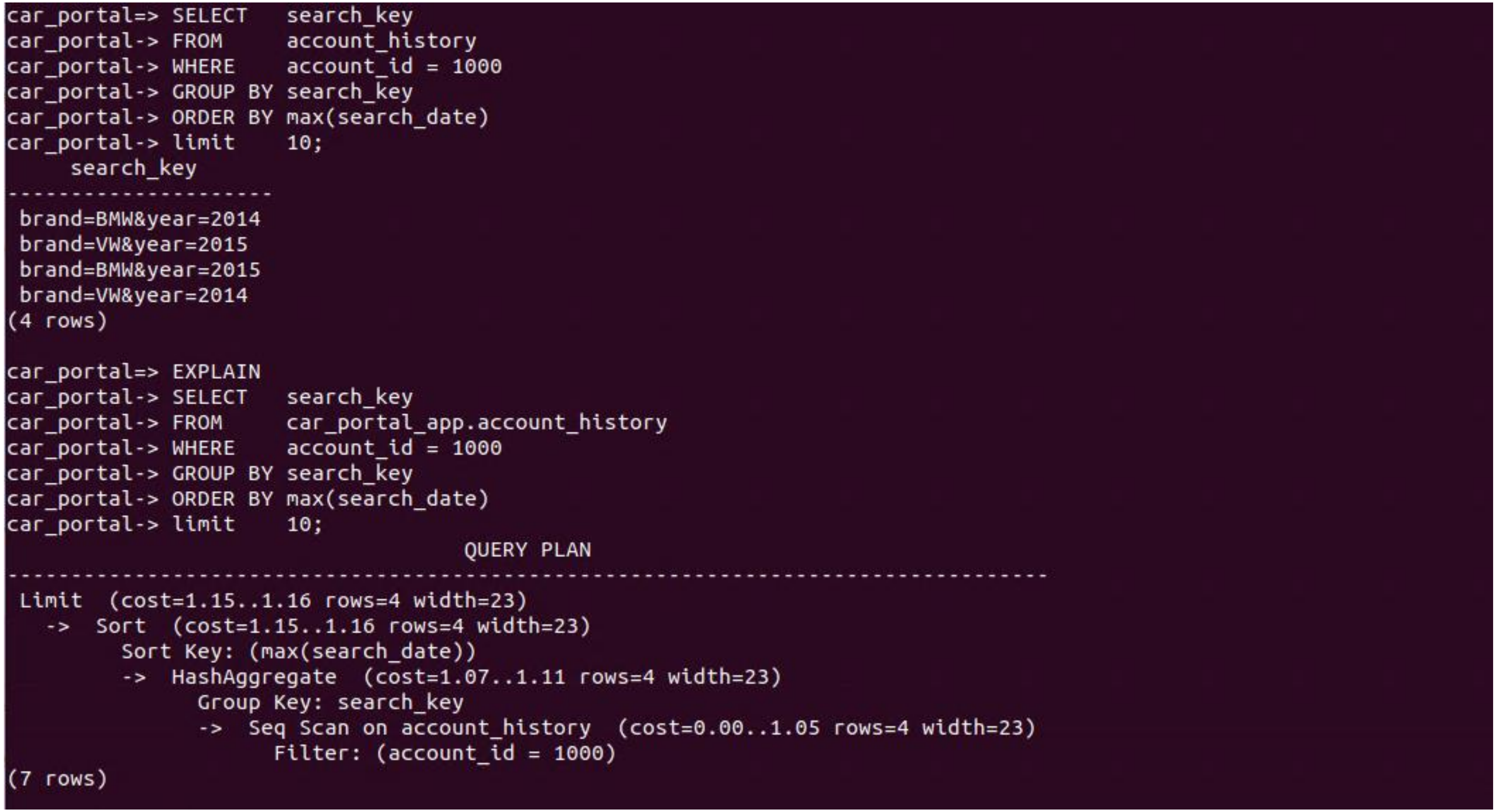
  -
    ```
    SELECT   search_key
    FROM     account_history
    WHERE    account_id = <account>
    GROUP BY search_key
    ORDER BY max(search_date) limit 10;
    ```
- The preceding query returns a maximum of ten records, containing different `search_key` instances, ordered by `search_date`.
- If we assume that the search `account_history` contains millions of rows, then reading all of the data would take a lot of time.
- In this case, this is not true, because the unique index will help us to read the data for only this particular customer.

- Indexes are not used on tables if the table size is small.
- The PostgreSQL planner will scan the complete table, instead.
- To show this, `account_history` was populated with a very small dataset, generated as follows:



```
car_portal=> WITH
car_portal-> test_account AS(
car_portal(>     INSERT INTO car_portal_app.account
car_portal(>     VALUES (1000, 'test_first_name', 'test_last_name','test3@email.com', 'password')
car_portal(>     RETURNING account_id),
car_portal-> car AS (
car_portal(>     SELECT i as car_model
car_portal(>     FROM (VALUES('brand=BMW'), ('brand=VW')) AS foo(i)),
car_portal-> manufacturing_date AS (
car_portal(>     SELECT 'year='|| i as date
car_portal(>     FROM generate_series (2015, 2014, -1) as foo(i))
car_portal-> INSERT INTO car_portal_app.account_history (account_id, search_key, search_date)
car_portal->     SELECT account_id, car.car_model||'&'||manufacturing_date.date, current_date
car_portal->     FROM test_account, car, manufacturing_date;
INSERT 0 4
car_portal=> VACUUM (VERBOSE, ANALYZE);
INFO:  vacuuming "car_portal_app.account"
INFO:  index "account_pkey" now contains 484 row versions in 4 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  index "account_email_key" now contains 484 row versions in 6 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
```

- To test whether the index is used, let's run the query, as follows:

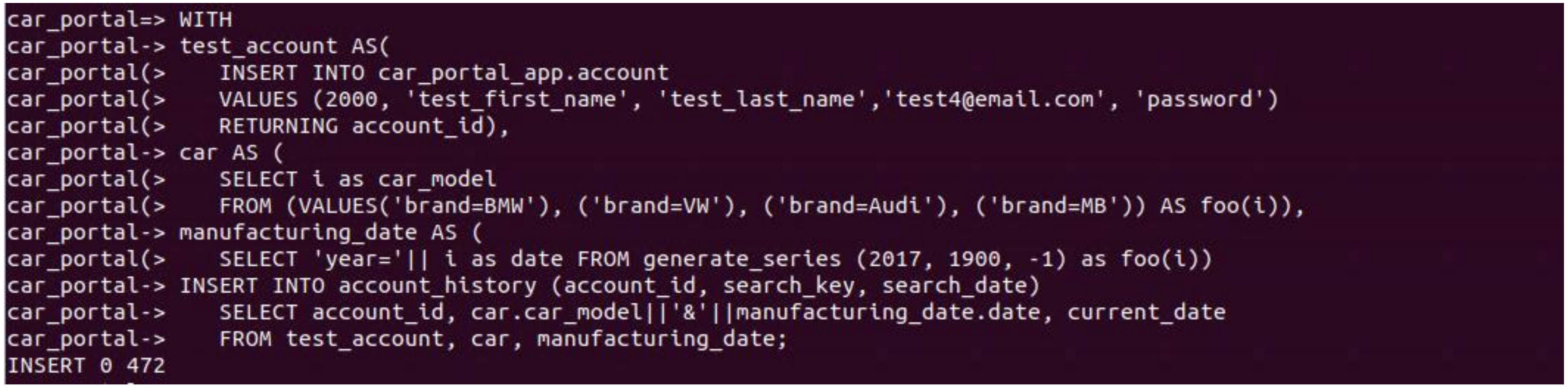


```
car_portal=> SELECT   search_key
car_portal-> FROM     account_history
car_portal-> WHERE    account_id = 1000
car_portal-> GROUP BY search_key
car_portal-> ORDER BY max(search_date)
car_portal-> limit    10;
     search_key
---------------------
 brand=BMW&year=2014
 brand=VW&year=2015
 brand=BMW&year=2015
 brand=VW&year=2014
(4 rows)

car_portal=> EXPLAIN
car_portal-> SELECT   search_key
car_portal-> FROM     car_portal_app.account_history
car_portal-> WHERE    account_id = 1000
car_portal-> GROUP BY search_key
car_portal-> ORDER BY max(search_date)
car_portal-> limit    10;
                                    QUERY PLAN
----------------------------------------------------------------------------------
 Limit  (cost=1.15..1.16 rows=4 width=23)
   ->  Sort  (cost=1.15..1.16 rows=4 width=23)
         Sort Key: (max(search_date))
         ->  HashAggregate  (cost=1.07..1.11 rows=4 width=23)
               Group Key: search_key
               ->  Seq Scan on account_history  (cost=0.00..1.05 rows=4 width=23)
                     Filter: (account_id = 1000)
(7 rows)
```

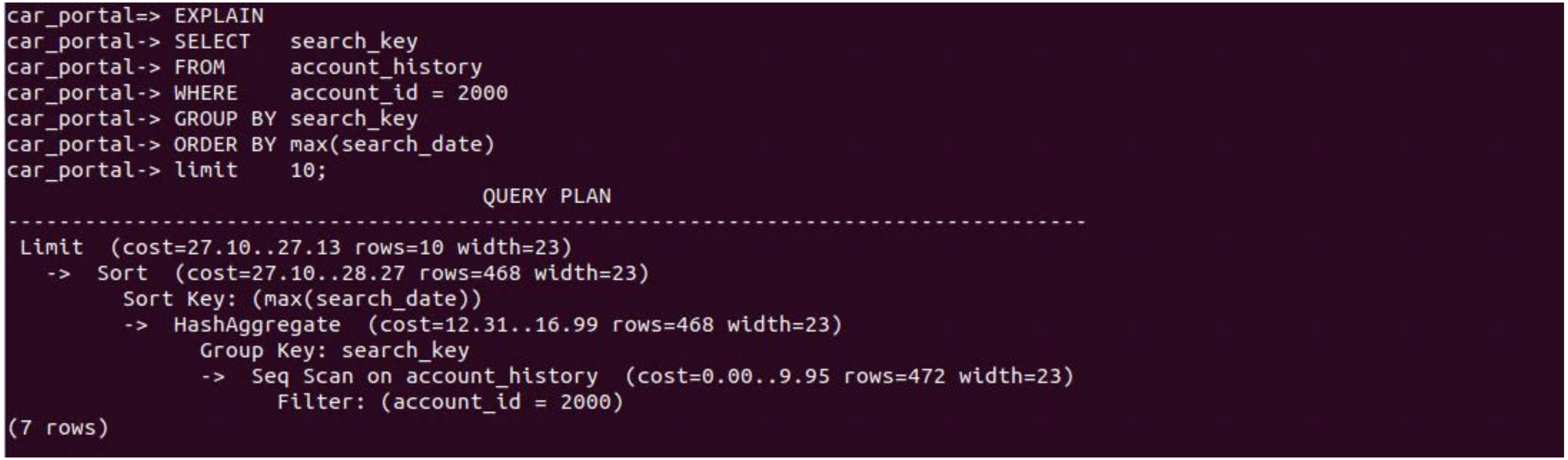
- The index in the preceding example is not used; the PostgreSQL planner decides whether to use an index based on the execution plan cost.
- For the same query with different parameters, the planner might pick a different plan, based on the data histogram.
- So, even if we have a big dataset and the predicate used in the query does not filter a lot of data, the index will not be used.

- To create such a scenario, let's insert some entries for another account and rerun the query, as follows:



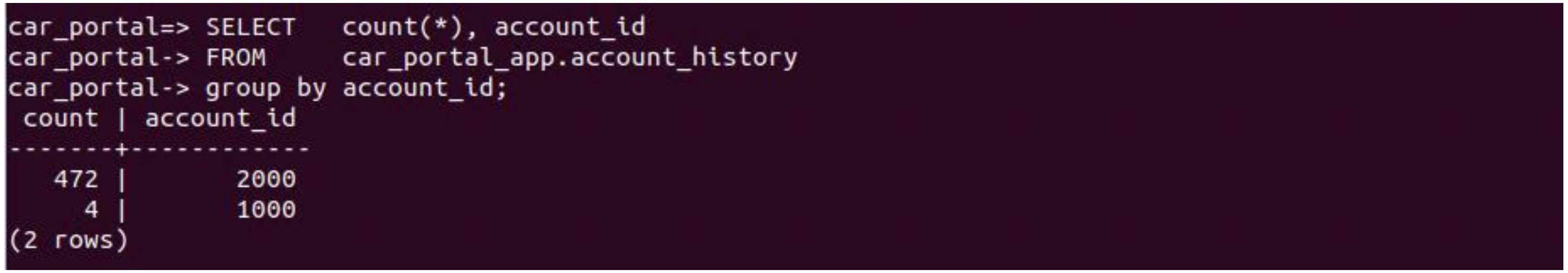
```
car_portal=> WITH
car_portal-> test_account AS(
car_portal(>    INSERT INTO car_portal_app.account
car_portal(>    VALUES (2000, 'test_first_name', 'test_last_name','test4@email.com', 'password')
car_portal(>    RETURNING account_id),
car_portal-> car AS (
car_portal(>    SELECT i as car_model
car_portal(>    FROM (VALUES('brand=BMW'), ('brand=VW'), ('brand=Audi'), ('brand=MB')) AS foo(i)),
car_portal-> manufacturing_date AS (
car_portal(>    SELECT 'year='|| i as date FROM generate_series (2017, 1900, -1) as foo(i))
car_portal-> INSERT INTO account_history (account_id, search_key, search_date)
car_portal->    SELECT account_id, car.car_model||'&'||manufacturing_date.date, current_date
car_portal->    FROM test_account, car, manufacturing_date;
INSERT 0 472
```

- To check whether the index will be chosen, let's run the query for the second account, which has more data, such as follows:



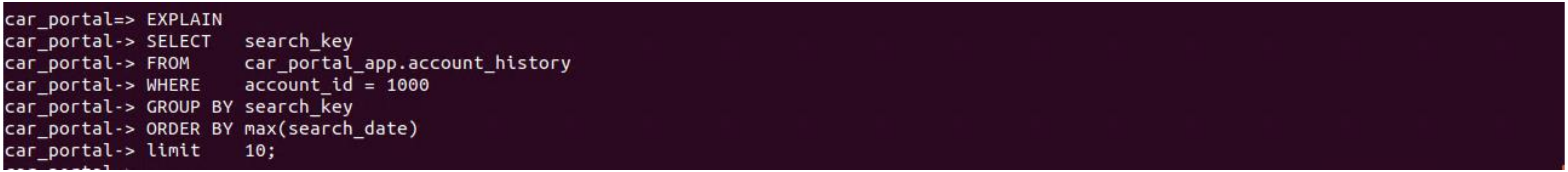
```
car_portal=> EXPLAIN
car_portal-> SELECT   search_key
car_portal-> FROM     account_history
car_portal-> WHERE    account_id = 2000
car_portal-> GROUP BY search_key
car_portal-> ORDER BY max(search_date)
car_portal-> limit    10;
                                  QUERY PLAN
--------------------------------------------------------------------------------------
 Limit  (cost=27.10..27.13 rows=10 width=23)
   ->  Sort  (cost=27.10..28.27 rows=468 width=23)
         Sort Key: (max(search_date))
         ->  HashAggregate  (cost=12.31..16.99 rows=468 width=23)
               Group Key: search_key
               ->  Seq Scan on account_history  (cost=0.00..9.95 rows=472 width=23)
                     Filter: (account_id = 2000)
(7 rows)
```

- Again, the index is not used, because we would like to read the whole table.
- The index selectivity here is very low, since account 2000 has 427 records, whereas account 1000 only has 4 records, as shown in the following snippet:

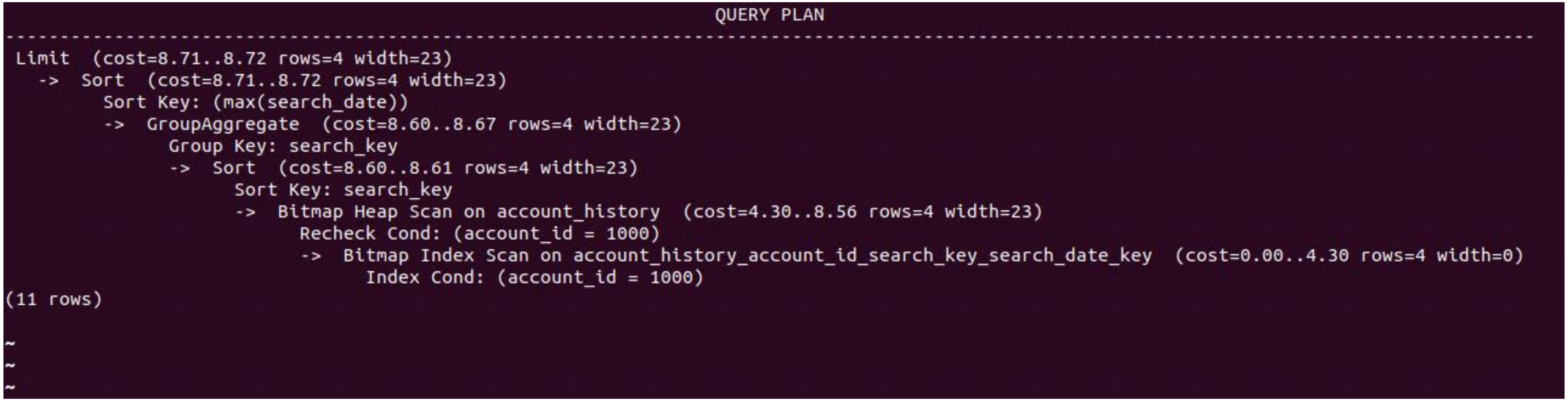


```
car_portal=> SELECT   count(*), account_id
car_portal-> FROM     car_portal_app.account_history
car_portal-> group by account_id;
 count | account_id
-------+------------
   472 |       2000
     4 |       1000
(2 rows)
```

- Finally, let's rerun the query for account 1000.
- In this case, the selectivity is high; hence, the index will be used:



```
car_portal=> EXPLAIN
car_portal-> SELECT   search_key
car_portal-> FROM     car_portal_app.account_history
car_portal-> WHERE    account_id = 1000
car_portal-> GROUP BY search_key
car_portal-> ORDER BY max(search_date)
car_portal-> limit    10;
```



```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=8.71..8.72 rows=4 width=23)
   ->  Sort  (cost=8.71..8.72 rows=4 width=23)
         Sort Key: (max(search_date))
         ->  GroupAggregate  (cost=8.60..8.67 rows=4 width=23)
               Group Key: search_key
               ->  Sort  (cost=8.60..8.61 rows=4 width=23)
                     Sort Key: search_key
                     ->  Bitmap Heap Scan on account_history  (cost=4.30..8.56 rows=4 width=23)
                           Recheck Cond: (account_id = 1000)
                           ->  Bitmap Index Scan on account_history_account_id_search_key_search_date_key  (cost=0.00..4.30 rows=4 width=0)
                                 Index Cond: (account_id = 1000)
(11 rows)

~
~
~
```

Index Types

- PostgreSQL supports different indexes; each index can be used for a certain scenario or data type, as follows:
  - **B-tree index**: This is the default index in PostgreSQL when the index type is not specified with the CREATE INDEX command. The **B** stands for **balanced**, which means that the data on both sides of the tree is roughly equal. A B-tree can be used for equality, ranges, and null predicates. The B-tree index supports all PostgreSQL data types.
  - **Hash index**: Prior to PostgreSQL 10, hash indexes are not well supported. They are not transaction-safe and are not replicated to the slave nodes in streaming replication. With PostgreSQL 10, the hash index limitations have been tackled. It is useful for equality predicates (=).

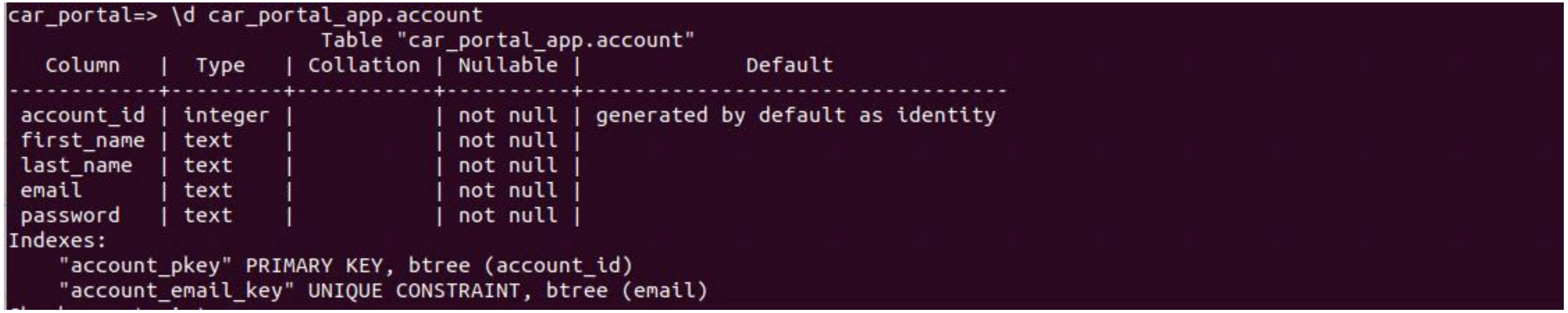
- **Generalized Inverted Index (GIN)**: The GIN index is useful when several values need to map to one row. It can be used with complex data structures, such as arrays and full-text searches.
- **Generalized Search Tree (GiST)**: The GiST indexes allow for the building of general balanced tree structures. They are useful in indexing geometric data types, as well as full-text searches.
- **Space-Partitioned GiST (SP-GiST)**: These are similar to GIST and support partitioned search trees. Generally speaking, `GIST`, `GIN`, and `SP-GIST` are designed to handle complex user-defined data types.
- **Block Range Index (BRIN)**: This was introduced in PostgreSQL 9.5. The BRIN index is useful for very large tables, where the size is limited. A BRIN index is slower than a B-tree index, but requires less space than the B-tree.

Index Categories

- We can classify indexes for several categories; note that these categories can be mixed up, which enables us to model very complex logic.
- For example, we could have a unique partial index, to make sure that only part of the data was unique, based on a certain condition.

- The index categories are as follows:
    - **Partial indexes**: A partial index only indexes a subset of the table data that meets a certain predicate; the `WHERE` clause is used with the index. The idea behind a partial index is to decrease the index size, making it more maintainable and faster to access.
    - **Covering indexes**: These were introduced in PostgreSQL 11. They allow some queries to be executed only using an index-only scan. Currently, this is only supported by B-tree indexes.

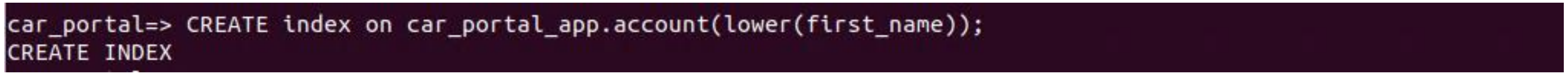
- **Unique indexes**: A unique index guarantees the uniqueness of a certain value in a row across the whole table. In the `account` table, the `email` column has a unique check constraint. This is implemented by the unique index, as shown by the `\d` meta-command:



```
car_portal=> \d car_portal_app.account
                        Table "car_portal_app.account"
   Column   |  Type   | Collation | Nullable |             Default
------------+---------+-----------+----------+----------------------------------
 account_id | integer |           | not null | generated by default as identity
 first_name | text    |           | not null |
 last_name  | text    |           | not null |
 email      | text    |           | not null |
 password   | text    |           | not null |
Indexes:
    "account_pkey" PRIMARY KEY, btree (account_id)
    "account_email_key" UNIQUE CONSTRAINT, btree (email)
```

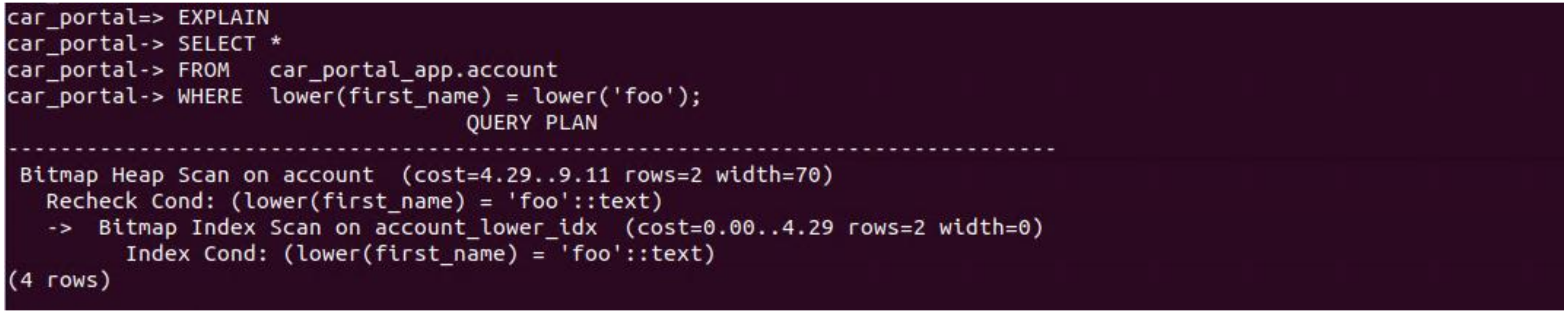
- **Multicolumn indexes**: These can be used for a certain pattern of query conditions. Suppose that a query has a pattern similar to the following: `SELECT * FROM table WHERE` $column_1$ `=` $constant_1$ `and` $column_2$`=` $constant_2$ `AND … ` $column_n$ `=` $constant_n$`;`. In this case, an index can be created on $column_1$, $column_2$,…, and $column_n$, where `n` is less than or equal to 32.
- **Indexes on expressions**: As we stated earlier, an index can be created on a table column or multiple columns. One can also be created on expressions and function results.

- For example, you can use indexes to optimize a case-insensitive search, by simply creating an index using the functions `lower()` or `upper()`, as follows:



```
car_portal=> CREATE index on car_portal_app.account(lower(first_name));
CREATE INDEX
```

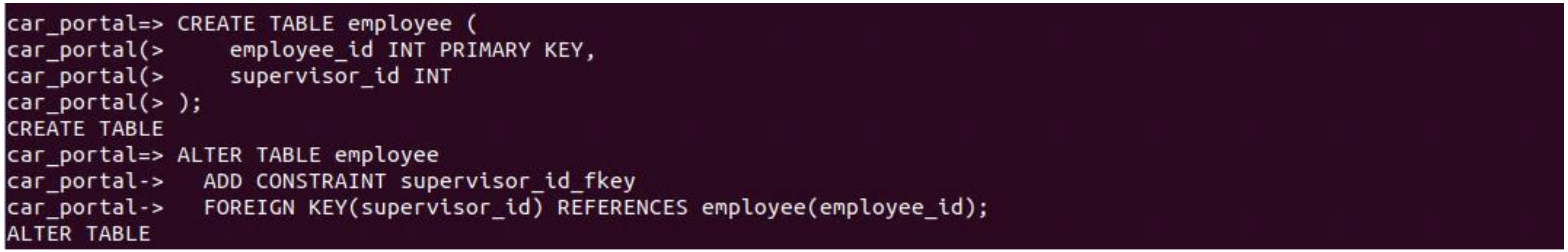
- The preceding index will speed up the performance of searching an account by names in a case-insensitive way, as follows:



```
car_portal=> EXPLAIN
car_portal-> SELECT *
car_portal-> FROM   car_portal_app.account
car_portal-> WHERE  lower(first_name) = lower('foo');
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on account  (cost=4.29..9.11 rows=2 width=70)
   Recheck Cond: (lower(first_name) = 'foo'::text)
   ->  Bitmap Index Scan on account_lower_idx  (cost=0.00..4.29 rows=2 width=0)
         Index Cond: (lower(first_name) = 'foo'::text)
(4 rows)
```

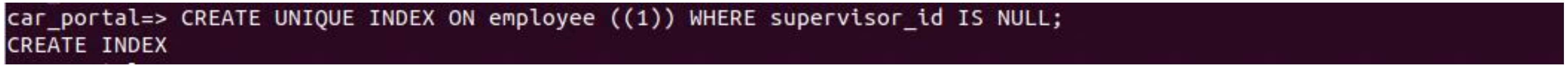
- Another use for expression indexes is to filter rows by casting a data type to another data type.
- For example, the departure time of a flight can be stored as `timestamp`; however, we often search for a `date` and not a time.

- As we mentioned earlier, you can also use unique and partial indexes together.
- For example, let's assume that we have a table called `employee`, where each employee must have a supervisor, except for the company head.
- We can model this as a self-referencing table, as follows:



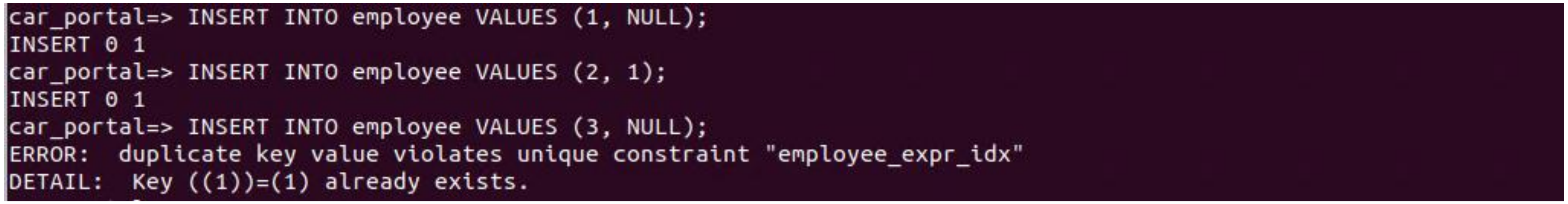
```
car_portal=> CREATE TABLE employee (
car_portal(>     employee_id INT PRIMARY KEY,
car_portal(>     supervisor_id INT
car_portal(> );
CREATE TABLE
car_portal=> ALTER TABLE employee
car_portal->   ADD CONSTRAINT supervisor_id_fkey
car_portal->   FOREIGN KEY(supervisor_id) REFERENCES employee(employee_id);
ALTER TABLE
```

- To guarantee that only one row is assigned to a supervisor, we can add the following unique index:



```
car_portal=> CREATE UNIQUE INDEX ON employee ((1)) WHERE supervisor_id IS NULL;
CREATE INDEX
```

- The unique index on the constant expression, `(1)`, will only allow one row with a null value.
- With the first insert of a null value, a unique index will be built, using the value `1`.
- A second attempt to add a row with a null value will cause an error, because the value `1` is already indexed:

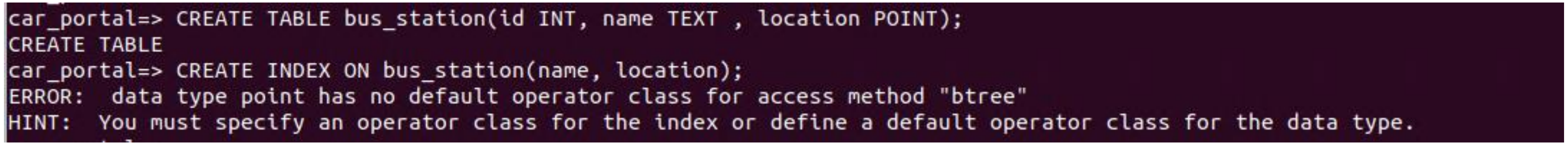


```
car_portal=> INSERT INTO employee VALUES (1, NULL);
INSERT 0 1
car_portal=> INSERT INTO employee VALUES (2, 1);
INSERT 0 1
car_portal=> INSERT INTO employee VALUES (3, NULL);
ERROR:  duplicate key value violates unique constraint "employee_expr_idx"
DETAIL:  Key ((1))=(1) already exists.
```

- Currently, only B-tree, GIN, GIST, and BRIN support multicolumn indexes.
- When creating a multicolumn index, the column order is important.
- Since the multicolumn index size is often big, the planner might prefer to perform a sequential scan, rather than reading the index.

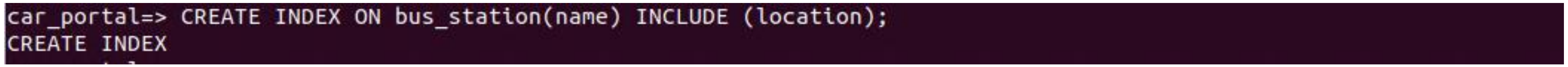
- Covering index is a new feature that was introduced in PostgreSQL 11; this feature allows the developer to use index-only scans.
- You can use multicolumn indexes to achieve index-only scans also, but the size of the index is a bit larger than a covering index.
- Also, the maintenance of multicolumn indexes is higher than for covering indexes.
- This feature is very useful in OLAP applications when the developer would like to have access to a certain set of columns using certain search criteria; so, in principle, it is similar to vertical partitioning.

- In addition, this feature can be used when multicolumn indexes cannot be used.
- PostgreSQL supports rich data types, and many of these data types do not have B-tree operators; see the following example:



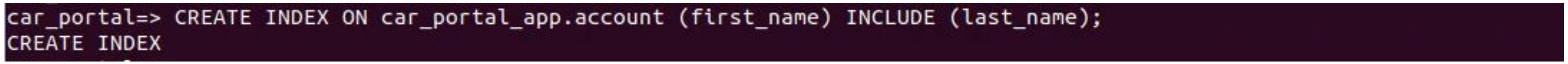
```
car_portal=> CREATE TABLE bus_station(id INT, name TEXT , location POINT);
CREATE TABLE
car_portal=> CREATE INDEX ON bus_station(name, location);
ERROR:  data type point has no default operator class for access method "btree"
HINT:  You must specify an operator class for the index or define a default operator class for the data type.
```

- If the developer is interested in retrieving the location of the bus station using an index-only scan when searching for bus stations by name, a solution could be as follows:



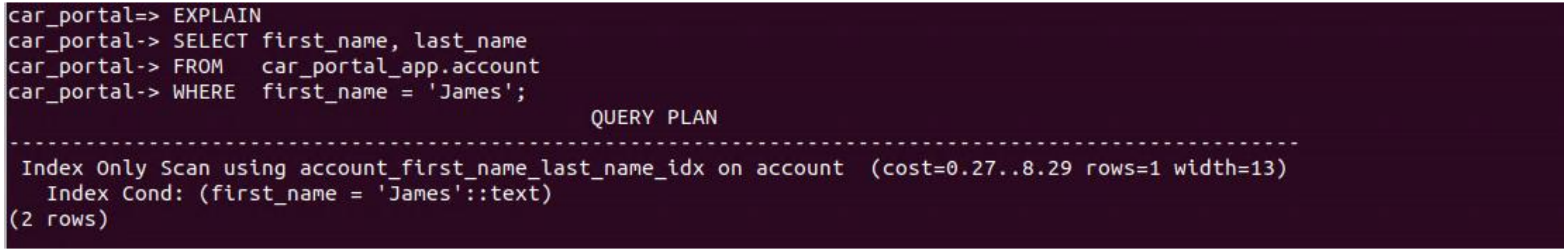
```
car_portal=> CREATE INDEX ON bus_station(name) INCLUDE (location);
CREATE INDEX
```

- To test this feature, let's create an index on `first_name`, which will be used as the search criteria, and include `last_name` in the index, as follows:



```
car_portal=> CREATE INDEX ON car_portal_app.account (first_name) INCLUDE (last_name);
CREATE INDEX
```

- The following example shows that an index-only scan is used to get `first_name` and `last_name` of a certain user:



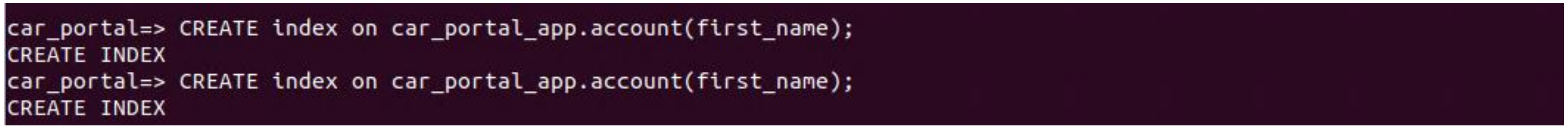
```
car_portal=> EXPLAIN
car_portal-> SELECT first_name, last_name
car_portal-> FROM   car_portal_app.account
car_portal-> WHERE  first_name = 'James';
                                               QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Index Only Scan using account_first_name_last_name_idx on account  (cost=0.27..8.29 rows=1 width=13)
   Index Cond: (first_name = 'James'::text)
(2 rows)
```

Best Practices for Indexes

- It is often useful to index columns that are used with predicates and foreign keys.
- This enables PostgreSQL to use an index scan instead of a sequential scan.
- The benefits of indexes are not limited to the `SELECT` statements; `DELETE` and `UPDATE` statements can also benefit from them.
- There are some cases where an index is not used, and this is often due to small table size.
- In big tables, we should plan the space capacity carefully, since the index size might be very big.
- Also, note that indexes have a negative impact on `INSERT` statements, since amending the index comes with a cost.

- There are several catalog tables and functions that help to maintain indexes, such as `pg_stat_all_indexes`, which gives statistics about index usage.

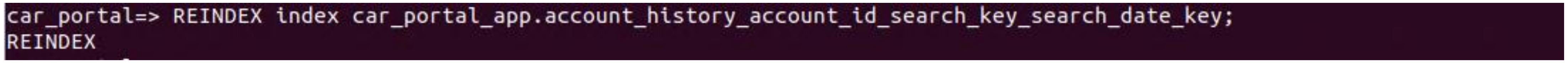
- When creating an index, make sure that the index does not exist; otherwise, you could end up with duplicate indexes.
- PostgreSQL does not raise a warning when duplicate indexes are created, as shown in the following example:



```
car_portal=> CREATE index on car_portal_app.account(first_name);
CREATE INDEX
car_portal=> CREATE index on car_portal_app.account(first_name);
CREATE INDEX
```

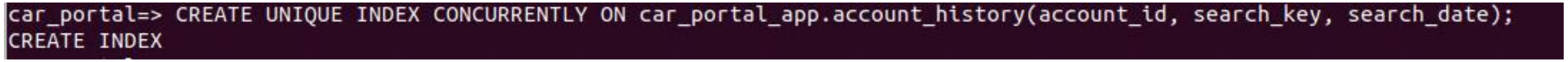
- In rare cases, an index might be bloated; PostgreSQL provides the `REINDEX` command, which is used to rebuild the index.
- Note that the `REINDEX` command is a blocking command.
- To solve this, you can concurrently create another index, identical to the original index, in order to avoid locks.
- Creating an index concurrently is often preferred in a live system, but it requires more work than creating a normal index.
- Moreover, creating an index concurrently has some caveats; in some cases, the index creation might fail, leading to an invalid index.
- An invalid index can be dropped or rebuilt by using the DROP and REINDEX statements, respectively.

- To reindex the index, `account_history_account_id_search_key_search_date_key`, you can run the following script. Note that this is a blocking statement:



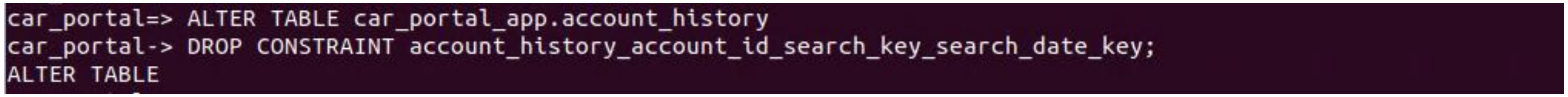
```
car_portal=> REINDEX index car_portal_app.account_history_account_id_search_key_search_date_key;
REINDEX
```

- Alternatively, you can create a concurrent index, which is not blocking, as follows:



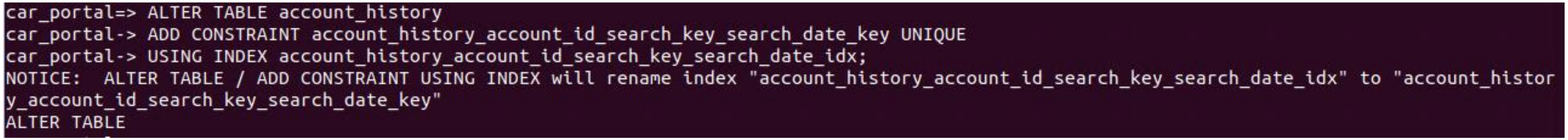
```
car_portal=> CREATE UNIQUE INDEX CONCURRENTLY ON car_portal_app.account_history(account_id, search_key, search_date);
CREATE INDEX
```

- Finally, to clean up, you will need to drop the original index.
- In this particular case, we cannot use the `DROP INDEX` statement, because the index was created via the unique constraint.
- To `DROP` the index, the constraint needs to be dropped, as follows:



```
car_portal=> ALTER TABLE car_portal_app.account_history
car_portal-> DROP CONSTRAINT account_history_account_id_search_key_search_date_key;
ALTER TABLE
```

- Finally, the unique constraint can be added, as follows:



```
car_portal=> ALTER TABLE account_history
car_portal-> ADD CONSTRAINT account_history_account_id_search_key_search_date_key UNIQUE
car_portal-> USING INDEX account_history_account_id_search_key_search_date_idx;
NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index "account_history_account_id_search_key_search_date_idx" to "account_histor
y_account_id_search_key_search_date_key"
ALTER TABLE
```