
Blind Index

Securely search encrypted database fields

Works with Lockbox (full example) and attr_encrypted (full example)

Learn more about securing sensitive data in Rails



How It Works

We use this approach by Scott Arciszewski. To summarize, we compute a keyed hash of the sensitive data and store it in a column. To query, we apply the keyed hash function to the value we're searching and then perform a database search. This results in performant queries for exact matches. Efficient `LIKE` queries are not possible, but you can index expressions.

Leakage

An important consideration in searchable encryption is leakage, which is information an attacker can gain. Blind indexing leaks that rows have the same value. If you use this for a field like last name, an attacker can use frequency analysis to predict the values. In an active attack where an attacker can control the input values, they can learn which other values in the database match.

Here's a great article on leakage in searchable encryption. Blind indexing has the same leakage as deterministic encryption.

Installation

Add this line to your application's Gemfile:

```
1 gem "blind_index"
```

Prep

Your model should already be set up with Lockbox or attr_encrypted. The examples are for a `User` model with `has_encrypted :email` or `attr_encrypted :email`. See the full examples for Lockbox and attr_encrypted if needed.

Also, if you use attr_encrypted, generate a key.

Getting Started

Create a migration to add a column for the blind index

```
1 add_column :users, :email_bidx, :string
2 add_index :users, :email_bidx # unique: true if needed
```

Add to your model

```
1 class User < ApplicationRecord
2   blind_index :email
3 end
```

For more sensitive fields, use

```
1 class User < ApplicationRecord
2   blind_index :email, slow: true
3 end
```

Backfill existing records

```
1 BlindIndex.backfill(User)
```

And query away

```
1 User.where(email: "test@example.org")
```

Expressions

You can apply expressions to attributes before indexing and searching. This gives you the the ability to perform case-insensitive searches and more.

```
1 class User < ApplicationRecord
2   blind_index :email, expression: ->(v) { v.downcase }
3 end
```

Validations

You can use blind indexes for uniqueness validations.

```
1 class User < ApplicationRecord
2   validates :email, uniqueness: true
3 end
```

We recommend adding a unique index to the blind index column through a database migration.

```
1 add_index :users, :email_bidx, unique: true
```

For `allow_blank: true`, use:

```
1 class User < ApplicationRecord
2   blind_index :email, expression: ->(v) { v.presence }
3   validates :email, uniqueness: {allow_blank: true}
4 end
```

For `case_sensitive: false`, use:

```
1 class User < ApplicationRecord
2   blind_index :email, expression: ->(v) { v.downcase }
3   validates :email, uniqueness: true # for best performance, leave out
    {case_sensitive: false}
4 end
```

Multiple Indexes

You may want multiple blind indexes for an attribute. To do this, add another column:

```
1 add_column :users, :email_ci_bidx, :string
2 add_index :users, :email_ci_bidx
```

Update your model

```
1 class User < ApplicationRecord
2   blind_index :email
3   blind_index :email_ci, attribute: :email, expression: ->(v) { v.
    downcase }
4 end
```

Backfill existing records

```
1 BlindIndex.backfill(User, columns: [:email_ci_bidx])
```

And query away

```
1 User.where(email_ci: "test@example.org")
```

Index Only

If you don't need to store the original value (for instance, when just checking duplicates), use a virtual attribute:

```
1 class User < ApplicationRecord
2   attribute :email, :string
3   blind_index :email
4 end
```

Multiple Columns

You can also use virtual attributes to index data from multiple columns:

```
1 class User < ApplicationRecord
2   attribute :initials, :string
3   blind_index :initials
4
5   before_validation :set_initials, if: -> { changes.key?(:first_name)
6     || changes.key?(:last_name) }
7
7   def set_initials
8     self.initials = "#{first_name[0]}#{last_name[0]}"
9   end
10 end
```

Migrating Data

If you're encrypting a column and adding a blind index at the same time, use the `migrating` option.

```
1 class User < ApplicationRecord
2   blind_index :email, migrating: true
3 end
```

This allows you to backfill records while still querying the unencrypted field.

```
1 BlindIndex.backfill(User)
```

Once that completes, you can remove the `migrating` option.

Key Rotation

To rotate keys without downtime, add a new column:

```
1 add_column :users, :email_bidx_v2, :string
2 add_index :users, :email_bidx_v2
```

And add to your model

```
1 class User < ApplicationRecord
2   blind_index :email, rotate: {version: 2, master_key: ENV["
  BLIND_INDEX_MASTER_KEY_V2"]}
3 end
```

This will keep the new column synced going forward. Next, backfill the data:

```
1 BlindIndex.backfill(User, columns: [:email_bidx_v2])
```

Then update your model

```
1 class User < ApplicationRecord
2   blind_index :email, version: 2, master_key: ENV["
  BLIND_INDEX_MASTER_KEY_V2"]
3 end
```

Finally, drop the old column.

Key Separation

The master key is used to generate unique keys for each blind index. This technique comes from CipherSweet. The table name and blind index column name are both used in this process.

You can get an individual key with:

```
1 BlindIndex.index_key(table: "users", bidx_attribute: "email_bidx")
```

To rename a table with blind indexes, use:

```
1 class User < ApplicationRecord
2   blind_index :email, key_table: "original_table"
3 end
```

To rename a blind index column, use:

```
1 class User < ApplicationRecord
2   blind_index :email, key_attribute: "original_column"
3 end
```

Algorithm

Argon2id is used for best security. The default cost parameters are 3 iterations and 4 MB of memory. For `slow: true`, the cost parameters are 4 iterations and 32 MB of memory.

A number of other algorithms are also supported. Unless you have specific reasons to use them, go with Argon2id.

Fixtures

You can use blind indexes in fixtures with:

```
1 test_user:
2   email_bidx: <%= User.generate_email_bidx("test@example.org").inspect
   %>
```

Be sure to include the `inspect` at the end or it won't be encoded properly in YAML.

Mongoid

For Mongoid, use:

```
1 class User
2   field :email_bidx, type: String
3   index({email_bidx: 1})
4 end
```

Key Generation

This is optional for Lockbox, as its master key is used by default.

Generate a key with:

```
1 BlindIndex.generate_key
```

Store the key with your other secrets. This is typically Rails credentials or an environment variable (dotenv is great for this). Be sure to use different keys in development and production. Keys don't need to be hex-encoded, but it's often easier to store them this way.

Set the following environment variable with your key (you can use this one in development)

```
1 BLIND_INDEX_MASTER_KEY=
   ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

or create `config/initializers/blind_index.rb` with something like

```
1 BlindIndex.master_key = Rails.application.credentials.
   blind_index_master_key
```

LIKE, ILIKE, and Full-Text Searching

Unfortunately, blind indexes can't be used for `LIKE`, `ILIKE`, or full-text searching. Instead, records must be loaded, decrypted, and searched in memory.

For `LIKE`, use:

```
1 User.select { |u| u.email.include?("value") }
```

For `ILIKE`, use:

```
1 User.select { |u| u.email =~ /value/i }
```

For full-text or fuzzy searching, use a gem like `FuzzyMatch`:

```
1 FuzzyMatch.new(User.all, read: :email).find("value")
```

If the number of records is large, try to find a way to narrow it down. An expression index is one way to do this, but leaks which records have the same value of the expression, so use it carefully.

Reference

Set default options in an initializer with:

```
1 BlindIndex.default_options = {algorithm: :pbkdf2_sha256}
```

By default, blind indexes are encoded in Base64. Set a different encoding with:

```
1 class User < ApplicationRecord
2   blind_index :email, encode: ->(v) { [v].pack("H*") }
3 end
```

By default, blind indexes are 32 bytes. Set a smaller size with:

```
1 class User < ApplicationRecord
2   blind_index :email, size: 16
3 end
```

Set a key directly for an index with:

```
1 class User < ApplicationRecord
2   blind_index :email, key: ENV["USER_EMAIL_BLIND_INDEX_KEY"]
3 end
```

Compatibility

You can generate blind indexes from other languages as well. For Python, you can use `argon2-cffi`.

```
1 from argon2.low_level import Type, hash_secret_raw
2 from base64 import b64encode
3
4 key = '289737bab72fa97b1f4b081cef00d7b7d75034bcf3183c363feaf3e6441777bc
5
6 value = 'test@example.org'
7
8 bidx = b64encode(hash_secret_raw(
9     secret=value.encode(),
10    salt=bytes.fromhex(key),
11    time_cost=3,
12    memory_cost=2**12,
13    parallelism=1,
14    hash_len=32,
15    type=Type.ID
16 ))
```

Alternatives

One alternative to blind indexing is to use a deterministic encryption scheme, like AES-SIV. In this approach, the encrypted data will be the same for matches. We recommend blind indexing over deterministic encryption because:

1. You can keep encryption consistent for all fields (both searchable and non-searchable)
2. Blind indexing supports expressions

Upgrading

2.0.0

2.0.0 brings a number of improvements.

- Blind indexes are updated immediately instead of in a `before_validation` callback
- Better Lockbox integration - no need to generate a separate key
- There's a new gem for Argon2 that has no dependencies and (officially) supports Windows

History

[View the changelog](#)

Contributing

Everyone is encouraged to help improve this project. Here are a few ways you can help:

- Report bugs
- Fix bugs and submit pull requests
- Write, clarify, or fix documentation
- Suggest or add new features

To get started with development and testing:

```
1 git clone https://github.com/ankane/blind_index.git
2 cd blind_index
3 bundle install
4 bundle exec rake test
```

For security issues, send an email to the address on this page.