# A6: Indices, triggers, user functions and population

SegFault is a collaborative platform for programmers to learn, discuss different approaches, present ideas and share knowledge in a Q&A style.

To this end, the following sections provide detailed insight into the inner workings of the project's database. The first section depicts the expected workload on the system, the second section specifies and explains the proposed indices to the database, and the third section comprises the database's triggers.

## 1. Database Workload

A study of the predicted system load (database load), organized in subsections.

### 1.1. Tuple Estimation

Estimate of tuples at each relation.

| Relation reference | Relation Name | Order of magnitude | Estimated growth |
|---|---|---|---|
| R01 | Table1 | units | dozens |
| R02 | Table2 | units | dozens |
| R03 | Table3 | units | dozens |
| R04 | Table4 | units | dozens |

### 1.2. Frequent Queries

Most important queries (SELECT) and their frequency.

| Query reference | SELECT01 | | Query description | One sentence describing the query goal | | Query frequency | magnitude per time | | ——————— | ——————————————————— | | SQL code |

### 1.3. Frequent Updates

Most important updates (INSERT, UPDATE, DELETE) and their frequency.

| Query reference | UPDATE01 | | Query description | One sentence describing the query goal | | Query frequency | magnitude per time | | ——————— | ——————————————————— | | SQL code |

## 2. Proposed Indices

This section presents the proposed indices on the database. It is important to note that many indices, mainly on high cardinality, would theoretically be better off being implemented as hash indices. We purposefuly did not choose these, because the PostgreSQL documentation actively discourages the usage of hash indices, as seen on the warning below.

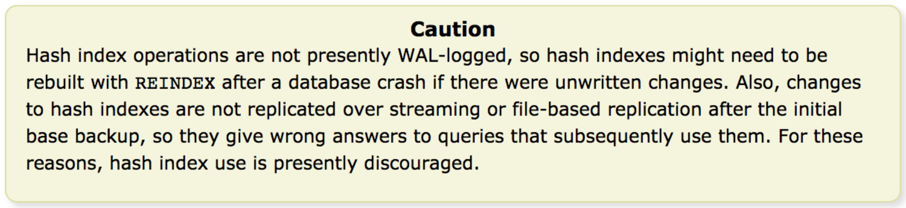
**Caution**
Hash index operations are not presently WAL-logged, so hash indexes might need to be rebuilt with `REINDEX` after a database crash if there were unwritten changes. Also, changes to hash indexes are not replicated over streaming or file-based replication after the initial base backup, so they give wrong answers to queries that subsequently use them. For these reasons, hash index use is presently discouraged.

Figure 1: Hash Indices - Caution

### 2.1. Performance Indices

| | |
|---|---|
| Index reference | IDX01 |
| Related queries | SELECT01 |
| Index relation | comment |
| Index attribute | commentable_id |
| Index type | B-tree |
| Cardinality | medium |
| Clustering | yes |
| Justification | The Table is very large, and query SELECT01 must run efficiently as it is executed several times. It doesn't need range query support, and is a good candidate for clustering as its cardinality is medium. |

```
CREATE INDEX comment_commentable ON comment USING btree(commentable_id);
```

(This index could be implemented as a hash index, but, as explained in this section's introduction, this is actively discouraged.)

| | |
|---|---|
| Index reference | IDX02 |
| Related queries | SELECT01, SELECT02, SELECT04, SELECT06, SELECT07, SELECT08, SELECT09, SELECT10 |
| Index relation | message_version |
| Index attribute | message_id |
| Index type | B-tree |
| Cardinality | medium |

| | |
|---|---|
| Clustering | yes |
| Justification | The Table is very large, and the corresponding queries are abundant and recurrent, thus must run efficiently. It doesn't need range query support, and is a good candidate for clustering as its cardinality is medium. |

```
CREATE INDEX message_version_message ON message_version USING btree(message_id);
```

| | |
|---|---|
| Index reference | IDX03 |
| Related queries | SELECT10 |
| Index relation | message |
| Index attribute | author |
| Index type | B-tree |
| Cardinality | medium |
| Clustering | yes |
| Justification | The Table is very large, and the corresponding queries are abundant and recurrent, thus must run efficiently. It doesn't need range query support, and is a good candidate for clustering as its cardinality is medium. |

```
CREATE INDEX message_author ON message USING btree(author);
```

| | |
|---|---|
| Index reference | IDX04 |
| Related queries | SELECT12 |
| Index relation | notification |
| Index attribute | user_id |
| Index type | B-tree |
| Cardinality | medium |
| Clustering | yes |
| Justification | The Table is very large, and the corresponding queries are abundant and recurrent, thus must run efficiently. It doesn't need range query support, and is a good candidate for clustering as its cardinality is medium. |

```
CREATE INDEX notification_user ON notification USING btree(user_id);
```

### 2.2. Full-text Search Indices

| Index reference | IDX05 | | Related queries | SELECT18 | | Index relation | tag | | Index attribute | name | | Index type | GIN | | Clustering | no | | Justification | To improve the performance of full text searches on the tag's name. GIN because the table is infrequently updated, and this type of indices takes longer to create/update but lead to faster lookups. No ts_vector encoding is used because the tag's name is just one word long, and it's lemma is rarely identifiable. |

```sql
CREATE INDEX tag_name ON tag USING gin(name);
```

| Index reference | IDX06 | | Related queries | SELECT19 | | Index relation | question | | Index attribute | title | | Index type | GiST | | Clustering | no | | Justification | To improve the performance of full text searches on the question's tile. GiST because it's better for dynamic data. |

```sql
CREATE INDEX question_title ON question USING gist(to_tsvector('english', title));
```

### 2.3. Contraint-enforcing Indices

The following indices are used to enforce special unique constraints, such as guaranteeing uniqueness of case insensitive usernames and emails.

```sql
CREATE INDEX unique_lowercase_username ON "user" (lower(username));
CREATE INDEX unique_lowercase_email ON "user" (lower(email));
```

## 3. Triggers

> User-defined functions and trigger procedures that add control structures to the SQL language or perform complex computations, are identified and described to be trusted by the database server. Every kind of function (SQL functions, Stored procedures, Trigger procedures) can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.

| Trigger reference | TRIGGER01 | | Trigger description | Trigger description, including reference to the business rules involved | | ————- | ————————————————— | | SQL code |

## 4. Complete SQL Code

> The database script must also include the SQL to populate a database with test data with an amount of tuples suitable for testing and with plausible values for the fields of the database. This code should also be included in the group's github repository as an SQL script, and a link include here.

## Revision history

Changes made to the first submission: 1. Item 1 1. Item 2

---

GROUP1763, 03/04/2018

André Cruz, up201503776@fe.up.pt
Daniel Marques, up201503822@fe.up.pt
Edgar Carneiro, up201503784@fe.up.pt
João Carvalho, up201504875@fe.up.pt