Database Optimization Report

1. Introduction

This report details six key optimizations identified in the project's Flask-SQLAlchemy models. The goal of these proposals is to enhance database performance, ensure long-term data integrity, and improve the overall efficiency, security, and maintainability of the application. The optimizations include a mix of schema-level changes and application-level recommendations.

2. Proposed Optimizations

2.1. High-Priority: Add Strategic Indexes

Problem:

Database lookups on non-primary keys (like foreign keys or status fields) require a "full table scan," which is extremely slow. As the `assignment`, `segment`, and `usr` tables grow, any query filtering by `segment_id`, `annotator_id`, or `status` will become a major performance bottleneck.

Solution:

Add an index (`index=True`) to all `db.ForeignKey` columns and any other columns frequently used in `WHERE` clauses.

Benefit:

This is the most critical optimization. It will make queries almost instantaneous by giving the database a sorted lookup table, dramatically improving application speed when loading related data (e.g., fetching all assignments for a user, or all segments for a chapter).

Example Code:

```
# In Assignment class
segment_id = db.Column(db.Integer, db.ForeignKey('segment.id'), nullable=False, index=True)
annotator_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False, index=True)
reviewer_id = db.Column(db.Integer, db.ForeignKey('user.id'), index=True)
annotation_status = db.Column(db.String(50), default='Unassigned', index=True) # Index this status field

# In USR class
segment_id = db.Column(db.Integer, db.ForeignKey('segment.id'), nullable=False, index=True)
created_by = db.Column(db.Integer, db.ForeignKey('user.id'), index=True)
status = db.Column(db.String(50), default='Pending', index=True) # Index this status field
```

2.2. Use Native PostgreSQL Enums for Status Fields

Problem:

Using `db.String` for columns with a fixed set of choices (e.g., `role`, `status`, `annotation_status`) is inefficient. It stores the full text string (e.g., "Unassigned") for every row, wastes space, and does not enforce data integrity (a typo like `"pendingg"` could be inserted).

Solution:

Replace `db.String` with PostgreSQL's native `db.Enum` type. This stores the value as a small, efficient integer internally but continues to validate inputs against a predefined list of strings.

Benefit:

- **Data Integrity:** Makes it impossible to insert invalid status values, preventing bugs.
- **Efficiency:** Reduces storage space and can speed up indexing on these columns.

Example Code:

```
import enum # Required import

# Define enums
class UserRoleEnum(enum.Enum):
    pending = 'pending'
    annotator = 'annotator'
    reviewer = 'reviewer'
    admin = 'admin'

class AnnotationStatusEnum(enum.Enum):
    Unassigned = 'Unassigned'
    Assigned = 'Assigned'
    InProgress = 'InProgress'
    Submitted = 'Submitted'

# Update models
class User(db.Model):
    # ...
    role = db.Column(db.Enum(UserRoleEnum), nullable=False, default=
   UserRoleEnum.pending, index=True)
    # ...

class Assignment(db.Model):
    # ...
    annotation_status = db.Column(db.Enum(AnnotationStatusEnum), default
   =AnnotationStatusEnum.Unassigned, index=True)
    # ...
```

2.3. Normalize Redundant Foreign Keys

Problem:

Several tables (`LexicalInfo`, `DependencyInfo`, `DiscourseCorefInfo`, `ConstructionInfo`, `SentenceTypeInfo`) store both a `usr_id` and a `segment_id`. This is redundant data because the `USR` model already has a `segment_id`. This redundancy wastes storage and can lead to data inconsistency if the IDs are mismatched.

Solution:

Remove the `segment_id` column and the corresponding `segment` relationship from all these child tables. The segment can always be accessed via the `usr` relationship (e.g., `lexical_info_object.usr.segment`).

Benefit:

- **Saves Storage:** Reduces the size of these frequently-used tables.
- **Ensures Data Consistency:** Eliminates the possibility of a `usr` and `segment` mismatch, making the data more robust.

Example Code (for `LexicalInfo`):

```
# BEFORE
class LexicalInfo(db.Model):
    # ...
    usr_id = db.Column(db.Integer, db.ForeignKey('usr.id'), nullable=False)
    segment_id = db.Column(db.Integer, db.ForeignKey('segment.id'), nullable=False) # <-- REDUNDANT
    usr = db.relationship('USR', back_populates='lexical_info')
    segment = db.relationship('Segment') # <-- REDUNDANT

# AFTER
class LexicalInfo(db.Model):
    # ...
    usr_id = db.Column(db.Integer, db.ForeignKey('usr.id'), nullable=False, index=True) # <-- Keep this
    usr = db.relationship('USR', back_populates='lexical_info')
    # No segment_id or segment relationship
```

2.4. Use GIN Index for `User.languages` Array

Problem:

The `User.languages` column uses a `db.ARRAY` type, which is a good choice. However, a standard database index is not effective for searching *inside* an array (e.g., "Find all users where `languages` contains 'hindi'"). This type of query will be slow.

Solution:

Add a specialized **GIN** (Generalized Inverted Index) to this column. GIN is a feature in PostgreSQL designed specifically for indexing the contents of complex types like arrays.

Benefit:

This will make searches for users by language extremely fast and efficient, improving any feature that involves filtering or finding annotators based on their language skills.

Example Code:

```
from sqlalchemy import Index # Add this import at the top

class User(db.Model):
    __tablename__ = 'user'
    # ... (all other columns) ...
    languages = db.Column(db.ARRAY(db.String(50)), nullable=False, default=['hindi'])

```

```
    # Add this at the end of the class
    __table_args__ = (
        Index('ix_user_languages_gin', languages, postgresql_using='gin'
    ),
    )
```

2.5. Fix `password_hash` Length (Bug Prevention)

Problem:

The `User.password_hash` column is currently limited to 200 characters. Modern, secure hashing algorithms (like `scrypt` or newer versions of `pbkdf2`) can produce hash strings longer than 200 characters. If a hash is longer, the database will cut it off (truncate it), and that user will be **permanently unable to log in**.

Solution:

Increase the length of this column from `String(200)` to `String(256)`.

Benefit:

This is a simple but critical bug-prevention fix. It ensures our application is compatible with modern and future security standards and prevents users from being locked out of their accounts.

Example Code:

```
class User(db.Model):
    # ...
    password_hash = db.Column(db.String(256), nullable=False)
    # ...
```