Progettazione database SemTUI

Introduzione

Si vuole progettare e realizzare di un database a supporto di SemTUI, un tool pensato per l'arricchimento semantico di dati tabellari.

Inizialmente, SemTUI si basava esclusivamente sull'utilizzo di file JSON per la memorizzazione di utenti, dataset e tabelle. Questa scelta, seppur rapida e flessibile nelle fasi iniziali di sviluppo, ha mostrato limiti evidenti in termini di scalabilità, consistenza, accesso concorrente e possibilità di effettuare query complesse.

La prima scelta implementativa riguardava la scelta tra un database relazionale (SQL) o per una soluzione NoSQL; i requisiti funzionali del sistema (quali, la necessità di effettuare query complesse con join, il mantenimento dell'integrità referenziale...) hanno motivato l'adozione di un database SQL. Tra le possibili alternative, la scelta è ricaduta su PostgreSQL soprattutto per il supporto avanzato per tipi JSONB, necessari per mantenere la flessibilità richiesta nel tipo di dati restituiti dalle operazioni di arricchimento, ma anche per le buone performance ottenute dal sistema di indicizzazione, e la semplicità d'integrazione con i microservizi Node.js utilizzati nel progetto.

Rendo disponibile link al repository contenente lo schema, le query presentate, le tabelle d'esempio per gli use cases e funzioni di supporto.

Schema

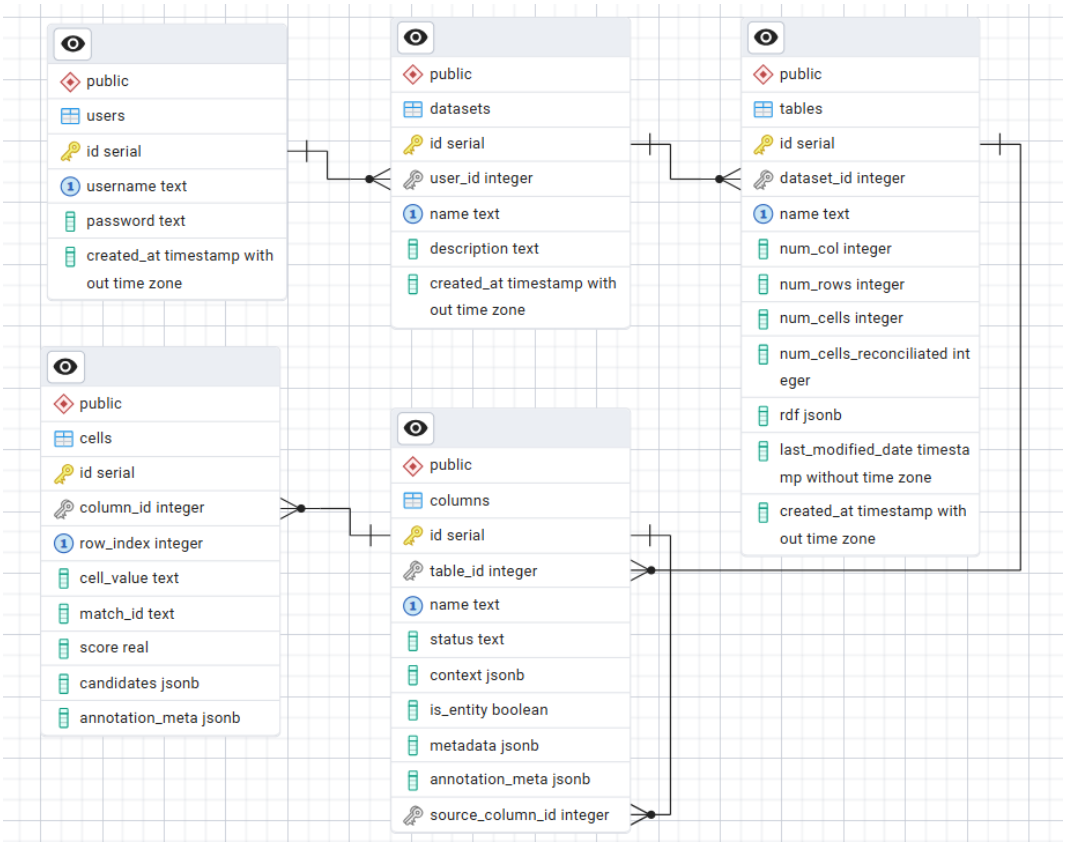

Tabella users

La tabella **users** rappresenta gli utenti registrati nel sistema. Ogni utente ha un identificativo univoco, un nome utente e una password che verrà poi cifrata; viene registrata anche la data di creazione dell'utente.

```sql
-- 1. Utenti
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username TEXT UNIQUE NOT NULL,
    password TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- **id**: identificativo univoco automatico
- **username**: nome univoco obbligatorio
- **password**: campo obbligatorio
- **created_at**: data di registrazione

Tabella datasets

Ogni utente può possedere più **dataset**. Ogni dataset è associato ad un utente e ha un nome univoco per quell'utente.

```
-- 2. Datasets di un utente
CREATE TABLE datasets (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    name TEXT NOT NULL,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, name)
);
```

- **id**: identificativo univoco automatico
- **user_id**: chiave esterna che fa riferimento all'id di un utente. L'eliminazione di un utente implica l'eliminazione di tutti i dataset associati.
- **name**: nome del dataset, campo obbligatorio
- **description**: descrizione opzionale del dataset
- **created_at**: data di creazione
- UNIQUE(user_id, name): lo stesso utente non può avere due dataset con lo stesso nome.

Tabella tables

Ogni dataset può contenere più **tabelle**. Ad ogni tabella è associato il dataset che la contiene e ha un nome univoco dentro a quel dataset. Ogni tabella ha associate una serie di informazioni strutturali e statistiche, quest'ultime aggiornate automaticamente.

```
-- 3. Tabelle contenute in un dataset
CREATE TABLE tables (
    id SERIAL PRIMARY KEY,
    dataset_id INTEGER REFERENCES datasets(id) ON DELETE CASCADE,
    name TEXT NOT NULL,
    num_col INTEGER NOT NULL,
    num_rows INTEGER NOT NULL,
    num_cells INTEGER NOT NULL,
    num_cells_reconciliated INTEGER NOT NULL,
    rdf JSONB DEFAULT '{}',
    last_modified_date TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(dataset_id, name)
);
```

- **id**: identificativo univoco automatico
- **name**: nome della tabella, campo obbligatorio
- **dataset**_id: chiave esterna che fa riferimento all'id di un dataset. L'eliminazione di un dataset implica l'eliminazione di tutte le tabelle associate.
- **num_col**, **num_rows**, **num_cells**, **num_cells_reconciliated**: valori inizialmente posti a 0 ed aggiornati automaticamente.

- **rdf**: campo di tipo JSONB per memorizzare, se presenti, le triple RDF associate ed i parametri di creazione. Ad ogni cambiamento effettuato su righe e colonne della tabella viene automaticamente ripristinato a '{}' poiché potrebbe essere invalido.
- **last_modified_date**: data dell'ultima modifica aggiornata automaticamente
- **created_at**: data di creazione
- UNIQUE(dataset_id, name): lo stesso dataset non può avere due tabelle con lo stesso nome

Tabella columns

Ogni tabella è composta da più **colonne**. Ad ogni colonna è associata la tabella a cui appartiene ed ha un nome univoco tra le colonne di quella tabella. Per ogni colonna non viene memorizzata solo la sua label ma, se presenti, anche informazioni ricavate dalle operazioni di riconciliazione ed estensione.

```
-- 4. Colonne per ciascuna tabella
CREATE TABLE columns (
    id SERIAL PRIMARY KEY,
    table_id INTEGER REFERENCES tables(id) ON DELETE CASCADE,
    name TEXT NOT NULL,
    status TEXT,
    context JSONB DEFAULT '{}',
    is_entity BOOLEAN DEFAULT FALSE,
    metadata JSONB DEFAULT '[]',
    annotation_meta JSONB DEFAULT '{}',
    source_column_id INTEGER REFERENCES columns(id),
    UNIQUE (table_id, name));
```

- **id**: identificativo univoco automatico
- **table_id**: chiave esterna che fa riferimento all'id della tabella di appartenenza. L'eliminazione di questa tabella implica l'eliminazione di tutte le sue colonne.
- **name**: nome della colonna, campo obbligatorio
- **status**: campo opzionale che indica lo stato della colonna (es. "reconciliated").
- **context**: campo opzionale che memorizza informazioni sul servizio utilizzato per le operazioni di arricchimento semantico
- **is_entity**: flag per distinguere le colonne che rappresentano entità
- metadata: array JSON che può contenere i metadati ricavati dalle operazioni di arricchimento. Può essere utile per avere più informazioni sulla colonna, quali tipo di dati rappresentato, proprietà che legano la colonna ad altre...
- **annotation_meta**: oggetto JSON che supporta ulteriori metadati di annotazione.
- **source_column_id**: chiave esterna opzionale, solo per colonne risultanti da operazioni di estensione, che fa riferimento all'id della colonna utilizzata per l'operazione
- UNIQUE(table_id, name): la stessa tabella non può avere due celle con lo stesso nome

Tabella cells

Ogni colonna ha un insieme di **celle**, una per ogni riga. Ad ogni cella la colonna a cui appartiene. Per ogni cella non viene memorizzata solo la sua label ma, se presenti, anche informazioni ricavate dalle operazioni di riconciliazione ed estensione.

```
-- 5. Celle della tabella
CREATE TABLE cells (
    id SERIAL PRIMARY KEY,
    column_id INTEGER REFERENCES columns(id) ON DELETE CASCADE,
    row_index INTEGER NOT NULL,
    cell_value TEXT,
    best_match_uri TEXT,
    best_match_label TEXT,
    score REAL,
    candidates JSONB DEFAULT '[]',
    annotation_meta JSONB DEFAULT '{}',
    UNIQUE (column_id, row_index)
);
```

- **id**: identificativo univoco automatico
- **column_id**: chiave esterna che fa riferimento all'id della colonna di appartenenza. L'eliminazione di questa colonna implica l'eliminazione di tutte le sue celle.
- **row_index**: indice obbligatorio rappresentante la riga della cella
- **cell_value**: contenuto della cella
- **best_match_uri**, **best_match_label**, **score**: memorizzazione delle informazioni del candidato che ha match con la cella, se presente
- **candidates**: array JSON con le possibili entità candidate
- **annotation_meta**: oggetto JSON che supporta ulteriori metadati di annotazione
- UNIQUE(column_id, row_index): univocità tra la coppia colonna (di una stessa tabella) ed indice di riga

Indici

```
-- 6. Indici
CREATE INDEX idx_datasets_user_id ON datasets(user_id);
CREATE INDEX idx_tables_dataset_id ON tables(dataset_id);
CREATE INDEX idx_columns_table_id ON columns(table_id);
CREATE INDEX idx_candidates_jsonb ON cells USING gin(candidates);
```

Crezione di indici per velocizzare le query:

- Dataset di un utente
- Tabelle di un dataset
- Colonne di una tabella
- Per la ricerca di celle di una tabella non occorre creare un index perché c'è già il vincolo unique
- Queries sui candidati di una cella

Triggers

Funzione per l'aggiornamento automatico nella tabella 'tables' dei dati statistici e ripristino del campo rdf:

```
-- 7. Trigger
CREATE OR REPLACE FUNCTION update_table_stats()
RETURNS TRIGGER AS $$
DECLARE
  col_table_id INTEGER;
BEGIN
  IF TG_TABLE_NAME = 'columns' THEN
    IF (TG_OP = 'DELETE') THEN
      col_table_id := OLD.table_id;
    ELSE
      col_table_id := COALESCE(NEW.table_id, OLD.table_id);
    END IF;
  ELSIF TG_TABLE_NAME = 'cells' THEN
    IF (TG_OP = 'DELETE') THEN
      SELECT table_id INTO col_table_id FROM columns WHERE id = OLD.column_id;
    ELSE
      SELECT table_id INTO col_table_id FROM columns WHERE id =
COALESCE(NEW.column_id, OLD.column_id);
    END IF;
  END IF;

  IF col_table_id IS NULL THEN
    RETURN NULL;
  END IF;

  UPDATE tables t
  SET
    num_col = (SELECT COUNT(*) FROM columns c WHERE c.table_id = col_table_id),
    num_rows = COALESCE((
      SELECT MAX(rr.row_index) + 1
      FROM cells rr
      JOIN columns c ON rr.column_id = c.id
      WHERE c.table_id = col_table_id
    ), 0),
    num_cells = (
      (SELECT COUNT(*) FROM columns c WHERE c.table_id = col_table_id) *
      COALESCE((
        SELECT MAX(rr.row_index) + 1
        FROM cells rr
        JOIN columns c ON rr.column_id = c.id
        WHERE c.table_id = col_table_id
      ), 0)
    ),
    num_cells_reconciliated = (
      SELECT COUNT(*)
```

```
      FROM cells rr
      JOIN columns c ON rr.column_id = c.id
      WHERE c.table_id = col_table_id AND c.status = 'reconciliated'
    ),
    rdf = '{}'::jsonb,
    last_modified_date = CURRENT_TIMESTAMP
  WHERE t.id = col_table_id;

  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Triggers per invocare la funzione ad ogni modifica sulle tabelle columns e cells:

```
CREATE TRIGGER trg_update_table_stats
AFTER INSERT OR UPDATE OR DELETE ON cells
FOR EACH ROW
EXECUTE FUNCTION update_table_stats();

CREATE TRIGGER trg_update_table_stats_on_columns
AFTER INSERT OR UPDATE OR DELETE ON columns
FOR EACH ROW
EXECUTE FUNCTION update_table_stats();
```

Query

Per facilitare l'integrabilità nel sistema originale, presento una serie di metodi per effettuare le query utili suddividendole in moduli node.js, un modulo per ogni tabella.

users.js

Funzionalità CRUD:

```
export async function createUser(username, password) {
  password = await bcrypt.hash(password, 10);
  const res = await pool.query(
    'INSERT INTO users (username, password) VALUES ($1, $2) RETURNING id, username,
created_at',
    [username, password]
  );
  return res.rows[0];
}
```

```
export async function getAllUsers() {
  const res = await pool.query('SELECT * FROM users');
  return res.rows;
}
```

```
export async function updateUser(id, newUsername, newPassword) {
  const hashed = await bcrypt.hash(newPassword, 10);
  const res = await pool.query(
```

```
    'UPDATE users SET username = $1, password = $2 WHERE id = $3 RETURNING id,
username, created_at',
    [newUsername, hashed, id]
  );
  return res.rows[0];
}
```

```
export async function deleteUser(id) {
  const res = await pool.query('DELETE FROM users WHERE id = $1 RETURNING id',
[id]);
  return res.rows[0];
}
```

Filtraggio dati:

```
export async function loginUser(username, password) {
  const res = await pool.query('SELECT * FROM users WHERE username = $1',
[username]);
  const user = res.rows[0];
  if (!user) return null;
  const isValid = await bcrypt.compare(password, user.password);
  if (!isValid) return null;
  return { id: user.id, username: user.username, created_at: user.created_at };
}

export async function getUserById(id) {
  const res = await pool.query('SELECT * FROM users WHERE id = $1', [id]);
  return res.rows[0];
}

export async function getIdByUser(username) {
  const res = await pool.query('SELECT id FROM users WHERE username = $1',
[username]);
  return res.rows[0]?.id;
}

export async function getUserByUsername(username) {
  const res = await pool.query('SELECT * FROM users WHERE username = $1',
[username]);
  return res.rows[0];
}
```

datasets.js

Funzionalià CRUD:

```
export async function createDataset(userId, name, description) {
  const res = await pool.query(
    'INSERT INTO datasets (user_id, name, description) VALUES ($1, $2, $3)
RETURNING *',
    [userId, name, description]
```

```
  );
  return res.rows[0];
}
```

```
export async function getAllDatasets() {
  const res = await pool.query('SELECT * FROM datasets');
  return res.rows;
}
```

```
export async function updateDataset(id, name, description) {
  const res = await pool.query(
    'UPDATE datasets SET name = $1, description = $2 WHERE id = $3 RETURNING *',
    [name, description, id]
  );
  return res.rows[0];
}
```

```
export async function deleteDataset(id) {
  const res = await pool.query('DELETE FROM datasets WHERE id = $1 RETURNING *',
[id]);
  return res.rows[0];
}
```

Filtraggio dati:

```
export async function getDatasetById(id) {
  const res = await pool.query('SELECT * FROM datasets WHERE id = $1', [id]);
  return res.rows;
}

export async function getDatasetsByUserId(userId, orderBy = 'name', order = 'ASC')
{
  const allowedOrderBy = ['name', 'id', 'created_at'];
  const allowedOrder = ['ASC', 'DESC'];
  const orderBySafe = allowedOrderBy.includes(orderBy) ? orderBy : 'name';
  const orderSafe = allowedOrder.includes(order.toUpperCase()) ?
order.toUpperCase() : 'ASC';

  const res = await pool.query(
    `SELECT * FROM datasets WHERE user_id = $1 ORDER BY ${orderBySafe}
${orderSafe}`,
    [userId]
  );
  return res.rows;
}

export async function getIdbyUserAndName(userId, datasetName) {
  const res = await pool.query(
    'SELECT id FROM datasets WHERE user_id = $1 AND LOWER(name) = LOWER($2)',
    [userId, datasetName]
  );
```

```
  return res.rows[0]?.id;
}

export async function getDatasetByName(name) {
  const res = await pool.query('SELECT * FROM datasets WHERE LOWER(name) LIKE
LOWER($1)', [`%${name}%`]);
  return res.rows;
}

export async function getDatasetByNameAndUser(userId, name) {
  const res = await pool.query(
    'SELECT * FROM datasets WHERE user_id = $1 AND LOWER(name) LIKE LOWER($2)',
    [userId, `%${name}%`]
  );
  return res.rows;
}
```

tables.js

Funzionalità CRUD:

```
export async function createTable(datasetId, name, nCols = 0, nRows = 0, nCells =
0, nCellsReconciliated = 0) {
  const res = await pool.query(
    `INSERT INTO tables
      (dataset_id, name, num_col, num_rows, num_cells, num_cells_reconciliated,
last_modified_date)
     VALUES ($1, $2, $3, $4, $5, $6, NOW()) RETURNING *`,
    [datasetId, name, nCols, nRows, nCells, nCellsReconciliated]
  );
  return res.rows[0];
}
```

```
export async function getAllTables() {
  const res = await pool.query('SELECT * FROM tables');
  return res.rows;
}
```

```
export async function updateTableName(id, newName) {
  const res = await pool.query(
    `UPDATE tables SET name = $1, last_modified_date = NOW() WHERE id = $2
RETURNING *`,
    [newName, id]
  );
  return res.rows[0];
}
```

```
export async function updateRDF(id, rdf){
  const res = await pool.query(
    `UPDATE tables SET rdf = $1, last_modified_date = NOW() WHERE id = $2 RETURNING
*`,
```

```
    [rdf, id]
  );
  return res.rows[0];
}
```

```
export async function deleteTable(id) {
  const res = await pool.query(
    `DELETE FROM tables WHERE id = $1 RETURNING *`,
    [id]);
  return res.rows[0];
}
```

Filtraggio dati:

```
export async function getTablesByDatasetId(datasetId) {
  const res = await pool.query(
    `SELECT *
     FROM tables WHERE dataset_id = $1 ORDER BY created_at DESC`,
    [datasetId]
  );
  return res.rows;
}

export async function getIdByDatasetAndName(datasetId, name) {
  const res = await pool.query(
    'SELECT id FROM tables WHERE dataset_id = $1 AND LOWER(name) = LOWER($2)',
    [datasetId, name]
  );
  return res.rows[0]?.id;
}

export async function getTableById(id) {
  const res = await pool.query('SELECT * FROM tables WHERE id = $1', [id]);
  return res.rows[0];
}

export async function searchTablesByName(name) {
  const res = await pool.query(
    'SELECT * FROM tables WHERE LOWER(name) LIKE LOWER($1)',
    [`%${name}%`]
  );
  return res.rows;
}

export async function searchTablesByUserAndName(userId, tableName) {
  const res = await pool.query(
    `SELECT t.*
     FROM tables t
     JOIN datasets d ON d.id = t.dataset_id
     WHERE d.user_id = $1 AND LOWER(t.name) LIKE LOWER($2)`,
```

```
    [userId, `%${tableName}%`]
  );
  return res.rows;
}
```

column.js

Funzionalità CRUD: occorre gestire le operazioni CRUD sia sulle colonne stesse (considerando sia colonne arricchite che non), sia sulle relazioni tra le colonne, indicate all'interno dell'array metadata tramite l'array 'property '.

La scelta di gestire le proprietà che legano le colonne tramite JSONB è dovuta alla grandissima flessibilità della rappresentazione delle relazioni tra colonne (assenti in molte tabelle) ed alle ottime performance delle query sui dati in JSONB, anche in presenza di molti dati.

```
export async function createColumn(tableId, name, status = null, context = {},
isEntity = false, metadata = [], annotationMeta = {}) {
  const res = await pool.query(
    `INSERT INTO columns
      (table_id, name, status, context, is_entity, metadata, annotation_meta)
     VALUES ($1, $2, $3, $4, $5, $6, $7) RETURNING *`,
    [tableId, name, status, JSON.stringify(context), isEntity,
JSON.stringify(metadata), JSON.stringify(annotationMeta)]
  );
  return res.rows[0];
}
```

```
export async function addPropertyToColumn(columnId, propertyObj) {
  const res = await pool.query(
    `
    UPDATE columns
    SET metadata = jsonb_set(
      metadata,
      '{0,property}',
      (COALESCE(metadata->0->'property', '[]'::jsonb) || $1::jsonb)
    )
    WHERE id = $2
    RETURNING *;
    `,
    [JSON.stringify(propertyObj), columnId]
  );
  return res.rows[0];
}
```

```
export async function updateColumnName(id, name) {
  const res = await pool.query(
    'UPDATE columns SET name = $1 WHERE id = $2 RETURNING *',
    [name, id]
  );
  return res.rows[0];
}
```

```
export async function updateReconciliationColumn(id, status = "reconciliated",
context = {}, isEntity = false, metadata = [], annotationMeta = {}) {
  const res = await pool.query(
    `UPDATE columns
     SET status = $1, context = $2, is_entity = $3, metadata = $4, annotation_meta
= $5
     WHERE id = $6 RETURNING *`,
    [status, JSON.stringify(context), isEntity, JSON.stringify(metadata),
JSON.stringify(annotationMeta),id]
  );
  return res.rows[0];
}
```

```
export async function deleteColumn(columnId) {
  const colRes = await pool.query(
    'SELECT name, table_id FROM columns WHERE id = $1',
    [columnId]
  );
  if (!colRes.rows.length) return null;
  const { name: columnName, table_id: tableId } = colRes.rows[0];

  await pool.query(
    `
    UPDATE columns
    SET metadata = jsonb_set(
      metadata,
      '{0,property}',
      COALESCE((
        SELECT jsonb_agg(prop)
        FROM jsonb_array_elements(metadata->0->'property') AS prop
        WHERE prop->>'obj' <> $1
      ), '[]'::jsonb)
    )
    WHERE table_id = $2
      AND id <> $3
      AND metadata->0->'property' IS NOT NULL
    `,
    [columnName, tableId, columnId]
  );

  const deleted = await pool.query(
    'DELETE FROM columns WHERE id = $1 RETURNING *',
    [columnId]
  );
  return deleted.rows[0];
}

export async function deletePropertyFromColumn(columnId, propertyId) {
  const res = await pool.query(
```

```
    `
    UPDATE columns
    SET metadata = jsonb_set(
      metadata,
      '{0,property}',
      COALESCE((
        SELECT jsonb_agg(prop)
        FROM jsonb_array_elements(metadata->0->'property') AS prop
        WHERE prop->>'id' <> $1
      ), '[]'::jsonb)
    )
    WHERE id = $2
    RETURNING *;
    `,
    [propertyId, columnId]
  );
  return res.rows[0];
}
```

NB: l'eliminazione di una colonna implica l'eliminazione delle relazioni con altre colonne, se presenti.

Operazioni di filtraggio dati:

```
export async function getIdByTableAndName(tableId, columnName) {
  const res = await pool.query(
    'SELECT id FROM columns WHERE table_id = $1 AND LOWER(name) = LOWER($2)',
    [tableId, columnName]
  );
  return res.rows[0]?.id;
}

export async function getColumnsByTableId(tableId) {
  const res = await pool.query('SELECT * FROM columns WHERE table_id = $1 ORDER BY
id', [tableId]);
  return res.rows;
}

export async function getColumnById(id) {
  const res = await pool.query('SELECT * FROM columns WHERE id = $1', [id]);
  return res.rows[0];
}

export async function getColumnByName(tableId, name) {
  const res = await pool.query(
    'SELECT * FROM columns WHERE table_id = $1 AND LOWER(name) = LOWER($2)',
    [tableId, name]
  );
  return res.rows[0];
}
```

```
export async function getReconciliatedColumns() {
  const res = await pool.query(
    `SELECT * FROM columns WHERE status = 'reconciliated' ORDER BY id`
  );
  return res.rows;
}

export async function getExtendedColumnsBySource(sourceColumnId) {
  const res = await pool.query(
    `
    SELECT *
    FROM columns
    WHERE source_column_id = $1
    ORDER BY id
    `,
    [sourceColumnId]
  );
  return res.rows;
}

export async function getPropertiesFromColumn(id) {
  const res = await pool.query(
    `
    SELECT metadata->0->'property' AS properties
    FROM columns
    WHERE id = $1
    `,
    [id]
  );
  return res.rows[0]?.properties || [];
}

export async function getAllPropertiesOfTable(tableId) {
  const res = await pool.query(
    `
    SELECT
      c1.id AS id_col1,
      c1.name AS name_col1,
      prop->>'id' AS id_property,
      c2.id AS id_col2,
      c2.name AS name_col2
    FROM columns c1
    JOIN LATERAL jsonb_array_elements(c1.metadata->0->'property') AS prop ON TRUE
    JOIN columns c2 ON prop->>'obj' = c2.name
    WHERE c1.metadata->0->'property' IS NOT NULL AND c1.table_id = $1
    `, [tableId]
  );
  return res.rows;
}
```

```
export async function getMetadataByColumnId(columnId) {
  const res = await pool.query(
    `
    SELECT metadata
    FROM columns
    WHERE id = $1
    `,
    [columnId]
  );
  if (!res.rows.length) return null;
  return res.rows[0].metadata;
}
```

cells.js

Funzionalità CRUD: occorre gestire le operazioni CRUD sia sui contenuti delle celle, sia sui candidati, mantenendo aggiornato il valore di match per ogni cella sia negli attributi della tabella, sia nei metadati.

```
export async function createCell(columnId, rowIndex, cellValue, bestMatchUri =
null, bestMatchLabel = null, score = null, candidates = [], annotationMeta = {}) {
  const res = await pool.query(
    `INSERT INTO cells
      (column_id, row_index, cell_value, best_match_uri, best_match_label, score,
candidates, annotation_meta)
     VALUES ($1, $2, $3, $4, $5, $6, $7, $8) RETURNING *`,
    [columnId, rowIndex, cellValue, bestMatchUri, bestMatchLabel, score,
JSON.stringify(candidates), JSON.stringify(annotationMeta)]
  );
  return res.rows[0];
}
```

```
export async function getAllResults() {
  const res = await pool.query('SELECT * FROM cells ORDER BY column_id,
row_index');
  return res.rows;
}
```

```
export async function updateCellLabel(id, cellValue) {
  const res = await pool.query(
    `UPDATE cells SET cell_value = $1 WHERE id = $2 RETURNING *`,
    [cellValue, id]
  );
  return res.rows[0];
}
```

```
export async function updateReconciliationResultById(id, bestMatchUri,
bestMatchLabel, score, candidates = [], annotationMeta = {}) {
  const res = await pool.query(
    `UPDATE cells SET best_match_uri = $1, best_match_label = $2, score = $3,
candidates = $4, annotation_meta = $5 WHERE id = $6 RETURNING *`,
```

```
    [bestMatchUri, bestMatchLabel, score, JSON.stringify(candidates),
JSON.stringify(annotationMeta), id]
  );
  return res.rows[0];
}
```

```
export async function updateReconciliationResultByColumnIdAndRow(columnId,
rowIndex, bestMatchUri, bestMatchLabel, score, candidates = [], annotationMeta =
{}) {
  const res = await pool.query(
    `UPDATE cells
     SET best_match_uri = $1, best_match_label = $2, score = $3, candidates = $4,
annotation_meta = $5
     WHERE column_id = $6 AND row_index = $7 RETURNING *`,
    [bestMatchUri, bestMatchLabel, score, JSON.stringify(candidates),
JSON.stringify(annotationMeta), columnId, rowIndex]
  );
  return res.rows[0];
}
```

```
export async function updateMatchById(id, matchUri, score) {
  const cellRes = await pool.query('SELECT candidates FROM cells WHERE id = $1',
[id]);
  if (!cellRes.rows.length) return null;
  let candidates = cellRes.rows[0].candidates;

  candidates = Array.isArray(candidates) ? candidates : [];

  let best = null;
  const rest = [];
  for (const cand of candidates) {
    if (
      cand.id === matchUri ||
      cand.name?.uri === matchUri
    ) {
      best = { ...cand, match: true, score: score }; // aggiorna score e match
    } else {
      rest.push({ ...cand, match: false });
    }
  }

  const newCandidates = best
    ? [best, ...rest]
    : [
        {
          id: matchUri,
          name: { uri: matchUri, value: '' },
          match: true,
          score: score
        },
```

```
        ...rest
      ];

  const res = await pool.query(
    `UPDATE cells
     SET match_id = $1, score = $2, candidates = $3
     WHERE id = $4 RETURNING *`,
    [matchUri, score, JSON.stringify(newCandidates), id]
  );
  return res.rows[0];
}
```

```
export async function getResultsByColumnId(columnId) {
  const res = await pool.query('SELECT * FROM cells WHERE column_id = $1 ORDER BY
row_index', [columnId]);
  return res.rows;
}

export async function getResultsByTableId(tableId) {
  const res = await pool.query('SELECT rr.* FROM cells rr JOIN columns c ON
rr.column_id = c.id WHERE c.table_id = $1 ORDER BY rr.row_index', [tableId]);
  return res.rows;
}

export async function getIdByColumnIdAndRow(columnId, rowIndex) {
  const res = await pool.query(
    'SELECT id FROM cells WHERE column_id = $1 AND row_index = $2',
    [columnId, rowIndex]
  );
  return res.rows[0]?.id;
}

export async function getResultById(id) {
  const res = await pool.query('SELECT * FROM cells WHERE id = $1', [id]);
  return res.rows[0];
}

export async function getMatchInfoByCellId(cellId) {
  const res = await pool.query(
    `
    SELECT cand AS match_info
    FROM cells,
         jsonb_array_elements(candidates) AS cand
    WHERE id = $1
      AND cand->>'id' = (SELECT match_id FROM cells WHERE id = $1)
    `,
    [cellId]
  );
  return res.rows[0]?.match_info || null;
```

```
}

export async function countResultsByColumnId(columnId) {
  const res = await pool.query(
    `SELECT COUNT(*) AS count FROM cells WHERE column_id = $1`,
    [columnId]
  );
  return parseInt(res.rows[0].count, 10);
}

export async function countResultsByTableId(tableId) {
  const res = await pool.query(
    `SELECT COUNT(*) AS count
     FROM cells rr
     JOIN columns c ON rr.column_id = c.id
     WHERE c.table_id = $1`,
    [tableId]
  );
  return parseInt(res.rows[0].count, 10);
}
export async function getResultsWithMinScore(score) {
  const res = await pool.query(
    `SELECT * FROM cells
     WHERE score >= $1`,
    [score]
  );
  return res.rows;
}

export async function getResultsWithMinScoreByColumnId(columnId, score) {
  const res = await pool.query(
    `SELECT * FROM cells
     WHERE column_id = $1 AND score >= $2`,
    [columnId, score]
  );
  return res.rows;
}

export async function getCandidatesByCellId(cellId) {
  const res = await pool.query(
    `SELECT rr.id AS cell_id,
            rr.column_id,
            rr.row_index,
            rr.cell_value,
            cand -> 'name' ->> 'uri'   AS candidate_uri,
            cand -> 'name' ->> 'value' AS candidate_label,
            (cand ->> 'score')::FLOAT AS candidate_score
    FROM cells rr,
         jsonb_array_elements(rr.candidates) AS cand
    WHERE jsonb_typeof(rr.candidates) = 'array' AND rr.id = $1`,
```

```
    [cellId]
  );
  return res.rows;
}

export async function getCandidatesWithMinScore(score) {
  const res = await pool.query(
    `SELECT
      rr.id AS cell_id,
      rr.column_id,
      rr.row_index,
      rr.cell_value,
      cand -> 'name' ->> 'uri'   AS candidate_uri,
      cand -> 'name' ->> 'value' AS candidate_label,
      (cand ->> 'score')::FLOAT AS candidate_score
    FROM cells rr,
         jsonb_array_elements(rr.candidates) AS cand
    WHERE jsonb_typeof(rr.candidates) = 'array' AND
    (cand ->> 'score')::FLOAT >= $1`,
    [score]
  );
  return res.rows;
}

export async function getCandidatesWithMinScoreByColumnId(columnId, score) {
  const res = await pool.query(
    `SELECT
      rr.id AS cell_id,
      rr.row_index,
      rr.cell_value,
      cand -> 'name' ->> 'uri'   AS candidate_uri,
      cand -> 'name' ->> 'value' AS candidate_label,
      (cand ->> 'score')::FLOAT AS candidate_score
    FROM cells rr,
         jsonb_array_elements(rr.candidates) AS cand
    WHERE jsonb_typeof(rr.candidates) = 'array' AND
    (cand ->> 'score')::FLOAT >= $1 AND rr.column_id = $2`,
    [score, columnId]
  );
  return res.rows;
}

export async function searchCellsByValuePrefix(prefix, columnId = null) {
  const params = [`${prefix}%`];
  let where = `rr.cell_value ILIKE $1`;
  if (columnId !== null) {
    where += ' AND rr.column_id = $2';
    params.push(columnId);
  }
  const res = await pool.query(
```

```
    `SELECT
      rr.id AS cell_id,
      rr.row_index,
      rr.cell_value,
      rr.column_id
    FROM cells rr
    WHERE ${where}`,
    params
  );
  return res.rows;
}

export async function searchCandidatesByLabelSubstring(substring, columnId = null)
{
  const params = [`%${substring}%`];
  let where = `jsonb_typeof(rr.candidates) = 'array' AND cand -> 'name' ->> 'value'
ILIKE $1`;
  if (columnId !== null) {
    where += ' AND rr.column_id = $2';
    params.push(columnId);
  }
  const res = await pool.query(
    `SELECT
      rr.id AS cell_id,
      rr.column_id,
      rr.row_index,
      rr.cell_value,
      cand -> 'name' ->> 'uri'   AS candidate_uri,
      cand -> 'name' ->> 'value' AS candidate_label,
      (cand ->> 'score')::FLOAT AS candidate_score
    FROM cells rr,
         jsonb_array_elements(rr.candidates) AS cand
    WHERE ${where}`,
    params
  );
  return res.rows;
}
```

Use case 1 – small table

Obiettivo: presentare lo schema utilizzato e alcuni output rilevanti nelle query utilizzando una tabella piccola in modo da facilitarne la comprensione.

La tabella di partenza è la seguente:

Country	Continent	Government Type	GDP Category	Capital	Currency Code	Independence Years
Italy	Europe	Republic	High	Rome	EUR	1861
France	Europe	Republic	High	Paris	EUR	843

Germany	Europe	Republic	High	Berlin	EUR	1871
Brazil	South America	Republic	Medium	Brazilia	BRL	1822
India	Asia	Republic	Medium	New Delhi	INR	1947
South Africa	Africa	Republic	Low	Pretoria	ZAR	1961

- La colonna 'Country' è stata utilizzata per la riconciliazione
- Le colonne 'Capital', 'Currency Code' e 'Independence Years' sono risultati da un'operazione di estensione dopo la riconciliazione

Query: performance e risultati

Per valutare le prestazioni del sistema, è stata implementata una funzione di test che carica la tabella in formato JSON (ho utilizzato il formato in cui venivano precedentemente memorizzati i dati aggiungendo, per questo caso d'uso, i risultati dell'operazione di estensione direttamente nel formato di risposta standardizzato), esegue automaticamente il parsing del file, popola le tabelle del database secondo la struttura definita e, infine, invoca le query precedentemente presentate, registrando i tempi di esecuzione per ciascuna operazione.

I risultati ottenuti sono riportati di seguito, compresi di visualizzazione dei risultati per alcune query.

I risultati comprendono sia i tempi d'esecuzione delle query, sia l'esecuzione di eventuali metodi di supporto.

1. Tempo creazione utente: 52.567ms
2. Tempo creazione dataset: 2.885ms
3. Tempo creazione tabella: 2.013ms
4. Tempo creazione colonne: 9.236ms
5. Tempo creazione celle: 19.243ms
6. Tempo creazione colonne estensione: 3.169ms
7. Tempo creazione celle estensione: 9.225ms

8. loginUser: 0.23ms
9. getUserByUsername: 0.578ms
10. getIdByUser: 0.292ms
11. getAllUsers: 0.691ms

	id [PK] integer	username text	password text	created_at timestamp without time zone
1	1	test_user	$2b$10$sFMrzFH5KiCoKmYdRJtM.edIWkqNqORA8EUXUeWm97aOEMRpcPl1O	2025-06-30 11:47:27.886888

12. getUserById: 0.712ms
13. updateUser: 50.87ms

14. updateDataset: 0.722ms

15. getAllDatasets: 0.352ms
16. getDatasetById: 0.381ms
17. getDatasetsByUserId: 0.495ms
18. getIdbyUserAndName: 0.295ms
19. getDatasetByName: 0.339ms
20. getDatasetByNameAndUser: 0.309ms

	id [PK] integer	user_id integer	name text	description text	created_at timestamp without time zone
1	1	1	Use case 1 dataset	Use case 1: small table	2025-06-30 11:47:27.892286

21. updateTableName: 0.671ms
22. updateRDF: 0.819ms
23. getAllTables: 0.335ms
24. getTablesByDatasetId: 0.368ms
25. getIdByDatasetAndName: 0.396ms
26. getTableById: 0.388ms
27. searchTablesByName: 0.371ms
28. searchTablesByUserAndName: 0.475ms

	id [PK] integer	dataset_id integer	name text	num_col integer	num_rows integer	num_cells integer	num_cells_reconciliated integer	rdf jsonb	last_modified_date timestamp without time zone	created_at timestamp without time zone
1	1	1	country_table	9	6	54	25	{}	2025-06-30 11:47:28.023214	2025-06-30 11:47:27.895947

29. createColumn: 0.911ms
30. createColumn: 0.884ms
31. updateColumnName: 1.061ms
32. addPropertyToColumn: 1.161ms
33. deletePropertyFromColumn: 1.328ms
34. getAllColumns: 0.473ms
35. getColumnsByTableId: 0.568ms

	id [PK] integer	table_id integer	name text	status text	context jsonb	is_entity boolean	metadata jsonb
1	1	1	Country	reconciliated	{"wd": {"uri": "https://www.wikidata.org/entity/", "total": 6, "prefix": "wd:", "reconciliated": 6}}	true	[{"id": "wd:Q38", "nam
2	2	1	Continent	empty	{}	false	[]
3	3	1	Government Type	empty	{}	false	[]
4	4	1	GDP Category	empty	{}	false	[]
5	5	1	capital	reconciliated	{}	true	[{"id": "capital", "name
6	6	1	currencyCode	reconciliated	{}	false	[{"id": "currencyCode"
7	7	1	independenceYears	reconciliated	{}	true	[{"id": "independence
8	8	1	test_column	reconciliated	{"contextKey": "contextValue"}	true	[{"id": "id1", "name": "
9	9	1	test_column2	empty	{}	false	[]

	annotation_meta jsonb	source_column_id integer
al", "name": "capital", "match": true, "score": "1"}]}	{"match": {"value": false}, "annotated": true, "lowestScore": 0, "highestScore": 0}	[null]
	{"match": {"value": false}, "annotated": false, "lowestScore": 0, "highestScore": 0}	[null]
	{"match": {"value": false}, "annotated": false, "lowestScore": 0, "highestScore": 0}	[null]
	{"match": {"value": false}, "annotated": false, "lowestScore": 0, "highestScore": 0}	[null]
	{}	1
	{}	1
	{}	1
	{}	[null]
	{}	[null]

36. getColumnById: 0.53ms
37. getColumnByName: 0.422ms
38. getIdByTableAndName: 0.321ms
39. getPropertiesFromColumn: 0.272ms
40. getAllPropertiesOfTable: 0.63ms

	id_col1 integer	name_col1 text	id_property text	id_col2 integer	name_col2 text
1	1	Country	wd:P36	5	capital
2	8	test_column	p1	9	test_column2

41. getMetadataByColumnId: 0.363ms
42. getExtendedColumnsBySource: 0.33ms

	id [PK] integer	table_id integer	name text	status text	context jsonb	is_entity boolean	metadata jsonb	an js
1	5	1	capital	reconciliated	{}	true	[{"id": "capital", "name": "Capital", "type": {"id": "entity", "name": "Entity"}, "service": "https://wikidata.org/api/reconcile"}]	{}
2	6	1	currencyCode	reconciliated	{}	false	[{"id": "currencyCode", "name": "Currency Code"}]	{}
3	7	1	independenceYears	reconciliated	{}	true	[{"id": "independenceYears", "name": "Independence Years", "type": {"id": "time_series", "name": "Time Series"}}]	{}

43. createCell: 0.735ms
44. updateCellLabel: 0.969ms
45. updateReconciliationResultById: 0.918ms
46. updateMatchById: 1.047ms
47. getAllResults: 0.506ms
48. getResultsByColumnId: 0.341ms
49. getResultsByTableId: 0.612ms

	id [PK] integer	column_id integer	row_index integer	cell_value text	match_id text	score real	candidates jsonb
1	1	1	0	Italy	wd:Q38	0.98	[{"id": "wd:Q38", "name": {"uri": "https
2	2	2	0	Europe	[null]	[null]	[]
3	3	3	0	Republic	[null]	[null]	[]
4	4	4	0	High	[null]	[null]	[]
5	25	5	0	Rome	Q220	[null]	[{"id": "Q220", "name": "Rome"}]
6	26	6	0	EUR	[null]	[null]	[{"str": "EUR"}]
7	27	7	0	1861	[null]	[null]	[{"str": "1861"}, {"str": "1946"}]
8	43	8	0	updated_cell	Q1	0.8	[{"id": "Q1", "name": "Italy", "match": t
9	5	1	1	France	wd:Q142	0.97	[{"id": "wd:Q142", "name": {"uri": "http
10	6	2	1	Europe	[null]	[null]	[]
11	7	3	1	Republic	[null]	[null]	[]
12	8	4	1	High	[null]	[null]	[]

50. getIdByColumnIdAndRow: 0.434ms
51. getResultById: 0.975ms
52. getMatchInfoByCellId: 0.951ms
53. countResultsByColumnId: 0.423ms
54. countResultsByTableId: 0.397ms
55. getResultsWithMinScore: 0.397ms
56. getResultsWithMinScoreByColumnId: 0.329ms
57. getCandidatesByCellId: 0.376ms

	cell_id integer	column_id integer	row_index integer	cell_value text	candidate_uri text	candidate_label text	candidate_score double precision
1	1	1	0	Italy	https://www.wikidata.org/wiki/Q38	Italy	0.98
2	1	1	0	Italy	https://www.wikidata.org/wiki/Q495	Italian Republic	0.85

58. getCandidatesWithMinScore: 0.472ms
59. getCandidatesWithMinScoreByColumnId: 0.505ms
60. searchCellsByValuePrefix: 0.509ms
61. searchCandidatesByLabelSubstring: 0.497ms

62. deleteColumn: 2.252ms
63. deleteTable: 1.412ms
64. deleteDataset: 0.662ms
65. deleteUser: 0.533ms

I tempi di esecuzione risultano molto contenuti per tutte le tipologie di query.

Use case 2: tabella con null

Obiettivo: in questo scenario si analizza il comportamento del sistema su una tabella priva di arricchimenti, che quindi presenta numerosi valori nulli nei metadati. Viene poi confrontata con la stessa tabella dopo l'applicazione delle operazioni di arricchimento.

Questa tabella è stata presa direttamente dai dati memorizzati nel tool, ed è una delle tabelle di dimensione maggiore: contiene 37 righe e 6 colonne.

Si vuole valutare:

- L'impatto dei valori null sulle prestazioni,
- Le differenze di struttura e dimensione tra la tabella originale e quella arricchita.

Query: performance e risultati

1. Tempo creazione utente: 98.696ms
2. Tempo creazione dataset: 4.778ms
3. Tempo creazione tabella: 7.707ms
4. Tempo creazione tabella: 1.738ms
5. Tempo creazione colonne: 81.335ms
6. Tempo creazione celle: 2.789s

7. loginUser: 0.469ms
8. getUserByUsername: 0.486ms
9. getIdByUser: 0.56ms
10. getAllUsers: 0.75ms
11. getUserById: 1.929ms
12. updateUser: 128.236ms

13. updateDataset: 1.36ms
14. getAllDatasets: 0.816ms
15. getDatasetById: 0.762ms
16. getDatasetsByUserId: 1.093ms
17. getIdbyUserAndName: 0.764ms
18. getDatasetByName: 1.622ms
19. getDatasetByNameAndUser: 1.199ms

20. updateTableName: 1.3ms
21. updateRDF: 1.204ms
22. getAllTables: 0.672ms
23. getTablesByDatasetId: 1.031ms
24. getIdByDatasetAndName: 0.578ms
25. getTableById: 0.599ms
26. searchTablesByName: 0.631ms
27. searchTablesByUserAndName: 0.816ms

28. createColumn: 6.186ms
29. createColumn: 3.126ms
30. updateColumnName: 2.858ms
31. addPropertyToColumn: 4.123ms

32. deletePropertyFromColumn: 4.89ms
33. getAllColumns: 0.954ms
34. getColumnsByTableId: 0.953ms
35. getColumnById: 0.724ms
36. getColumnByName: 0.801ms
37. getIdByTableAndName: 1.194ms
38. getReconciliatedColumnsByTableId: 1.251ms
39. getPropertiesFromColumn: 0.69ms
40. getAllPropertiesOfTable: 1.18ms
41. getMetadataByColumnId: 0.65ms
42. getExtendedColumnsBySource: 0.603ms

43. createCell: 2.879ms
44. updateCellLabel: 7.889ms
45. updateReconciliationResultById: 3.001ms
46. updateMatchById: 2.938ms
47. getAllResults: 22.61ms
48. getResultsByColumnId: 1.256ms
49. getResultsByTableId: 7.776ms
50. getIdByColumnIdAndRow: 1.352ms
51. getResultById: 1.427ms
52. getMatchInfoByCellId: 1.191ms
53. countResultsByColumnId: 0.91ms
54. countResultsByTableId: 1.317ms
55. getResultsWithMinScore: 0.867ms
56. getResultsWithMinScoreByColumnId: 0.592ms
57. getCandidatesByCellId: 0.734ms
58. getCandidatesWithMinScore: 3.339ms
59. getCandidatesWithMinScoreByColumnId: 0.838ms
60. searchCellsByValuePrefix: 1.672ms
61. searchCandidatesByLabelSubstring: 1.481ms

62. deleteColumn: 5.037ms
63. deleteTable: 25.241ms
64. deleteDataset: 1.361ms
65. deleteUser: 1.121ms

Tabella estesa

1. Tempo creazione utente: 118.57ms
2. Tempo creazione dataset: 1.334ms
3. Tempo creazione tabella: 1.571ms
4. Tempo creazione tabella: 1.837ms

5. Tempo creazione tabella: 1.345ms
6. Tempo creazione tabella: 2.421ms
7. Tempo creazione colonne: 16.037ms
8. Tempo creazione celle: 283.667ms
9. loginUser: 0.477ms
10. getUserByUsername: 0.484ms
11. getIdByUser: 0.519ms
12. getAllUsers: 0.468ms
13. getUserById: 0.524ms
14. updateUser: 99.25ms
15. updateDataset: 1.067ms
16. getAllDatasets: 0.517ms
17. getDatasetById: 0.66ms
18. getDatasetsByUserId: 0.64ms
19. getIdbyUserAndName: 0.581ms
20. getDatasetByName: 0.692ms
21. getDatasetByNameAndUser: 1.096ms
22. updateTableName: 1.741ms
23. updateRDF: 1.323ms
24. getAllTables: 0.807ms
25. getTablesByDatasetId: 2.071ms
26. getIdByDatasetAndName: 0.95ms
27. getTableById: 0.969ms
28. searchTablesByName: 0.851ms
29. searchTablesByUserAndName: 0.975ms
30. createColumn: 1.57ms
31. createColumn: 1.611ms
32. updateColumnName: 1.899ms
33. addPropertyToColumn: 2.714ms
34. deletePropertyFromColumn: 2.05ms
35. getAllColumns: 0.902ms
36. getColumnsByTableId: 0.901ms
37. getColumnById: 0.841ms
38. getColumnByName: 1.011ms
39. getIdByTableAndName: 0.877ms
40. getReconciliatedColumnsByTableId: 1.075ms
41. getPropertiesFromColumn: 0.985ms
42. getAllPropertiesOfTable: 1.068ms
43. getMetadataByColumnId: 0.947ms
44. getExtendedColumnsBySource: 0.605ms
45. createCell: 1.867ms
46. updateCellLabel: 1.818ms

47. updateReconciliationResultById: 2.282ms
48. updateMatchById: 3.017ms
49. getAllResults: 8.036ms
50. getResultsByColumnId: 4.88ms
51. getResultsByTableId: 6.579ms
52. getIdByColumnIdAndRow: 1.238ms
53. getResultById: 0.656ms
54. getMatchInfoByCellId: 0.771ms
55. countResultsByColumnId: 0.622ms
56. countResultsByTableId: 1.096ms
57. getResultsWithMinScore: 3.457ms
58. getResultsWithMinScoreByColumnId: 2.775ms
59. getCandidatesByCellId: 0.991ms
60. getCandidatesWithMinScore: 2.944ms
61. getCandidatesWithMinScoreByColumnId: 2.218ms
62. searchCellsByValuePrefix: 1.041ms
63. searchCandidatesByLabelSubstring: 1.265ms
64. deleteColumn: 7.135ms
65. deleteTable: 4.062ms
66. deleteDataset: 1.538ms
67. deleteUser: 1.19ms

Il sistema ha ottime prestazioni in entrambi i casi.

Use case 3: a bigger table

Obiettivo: in quest'ultimo scenario si effettuano test sul comportamento del sistema su una tabella contenente 10.000 righe. A differenza delle altre, questa tabella non è direttamente presente nel repository, ma è presente lo script per generare questa nuova tabella estendendo la tabella utilizzata nello use case precedente.

Query: performance e risultati

1. Tempo creazione utente: 113.978ms
2. Tempo creazione dataset: 4.616ms
3. Tempo creazione tabella: 7.122ms
4. Tempo creazione tabella: 1.559ms
5. Tempo creazione tabella: 1.565ms
6. Tempo creazione tabella: 1.583ms
7. Tempo creazione colonne: 58.58ms
8. Tempo creazione celle: 896513 ms
9. loginUser: 1.105ms
10. getUserByUsername: 0.755ms

11. getIdByUser: 0.608ms
12. getAllUsers: 0.855ms
13. getUserById: 0.859ms
14. updateUser: 111.794ms
15. updateDataset: 1.807ms
16. getAllDatasets: 0.824ms
17. getDatasetById: 0.898ms
18. getDatasetsByUserId: 1.337ms
19. getIdbyUserAndName: 1.05ms
20. getDatasetByName: 1.121ms
21. getDatasetByNameAndUser: 0.908ms
22. updateTableName: 1.626ms
23. updateRDF: 1.977ms
24. getAllTables: 1.356ms
25. getTablesByDatasetId: 1.268ms
26. getIdByDatasetAndName: 0.748ms
27. getTableById: 0.979ms
28. searchTablesByName: 1.041ms
29. searchTablesByUserAndName: 1.25ms
30. createColumn: 35.659ms
31. createColumn: 31.226ms
32. updateColumnName: 31.306ms
33. addPropertyToColumn: 33.396ms
34. deletePropertyFromColumn: 31.911ms
35. getAllColumns: 1.447ms
36. getColumnsByTableId: 1.634ms
37. getColumnById: 1.058ms
38. getColumnByName: 1.094ms
39. getIdByTableAndName: 0.92ms
40. getReconciliatedColumnsByTableId: 1.067ms
41. getPropertiesFromColumn: 0.925ms
42. getAllPropertiesOfTable: 1.593ms
43. getMetadataByColumnId: 1.133ms
44. getExtendedColumnsBySource: 1.194ms
45. createCell: 30.903ms
46. updateCellLabel: 32.187ms
47. updateReconciliationResultById: 31.006ms
48. updateMatchById: 35.123ms
49. getAllResults: 344.046ms
50. getResultsByColumnId: 125.101ms
51. getResultsByTableId: 445.805ms
52. getIdByColumnIdAndRow: 6.48ms

53. getResultById: 1.391ms
54. getMatchInfoByCellId: 1.479ms
55. countResultsByColumnId: 2.933ms
56. countResultsByTableId: 15.217ms
57. getResultsWithMinScore: 125.938ms
58. getResultsWithMinScoreByColumnId: 44.292ms
59. getCandidatesByCellId: 15.21ms
60. getCandidatesWithMinScore: 212.643ms
61. getCandidatesWithMinScoreByColumnId: 67.374ms
62. searchCellsByValuePrefix: 17.974ms
63. searchCandidatesByLabelSubstring: 67.589ms
64. deleteColumn: 121.9ms
65. deleteTable: 748.143ms
66. deleteDataset: 1.754ms
67. deleteUser: 1.502ms

I risultati mostrano una grande criticità nei tempi di scrittura per una tabella così grande, arrivando a metterci parecchi minuti, tuttavia le query risultano molto rapide.

Conclusioni

L'architettura complessiva del sistema e le relative scelte progettuali rappresentano un buon compromesso tra flessibilità del modello dati e prestazioni operative.

Il progetto implementa tutte le principali tipologie di query richieste, comprese operazioni di CRUD, filtraggio e ordinamento, ricerca testuale e query su campi annidati in JSONB, dimostrandosi efficiente sia nell'elaborazione di dati tabellari tradizionali, sia nella gestione di contenuti arricchiti con metadati semantici memorizzati in formato JSONB.

L'utilizzo di indici ha significativamente le performance delle interrogazioni, anche in presenza di metadati complessi. La struttura gerarchica del modello e l'utilizzo di trigger per l'aggiornamento automatico dei campi derivati, ha contribuito a garantire e facilitare la consistenza dei dati.

Il maggior limite riscontrato riguarda l'inserimento massivo di grandi volumi di dati, che risulta molto lento, sarebbe quindi opportuno introdurre tecniche di parallelizzazione. Tuttavia, per carichi di dati medi, il sistema si comporta in modo più che soddisfacente.

Infine, la scelta di utilizzare un'architettura modulare dovrebbe rendere l'integrazione con il backend Node.js di SemTUI molto semplice.