# Data Management

import tables PostgreSQL:

- Badges: 12,78 sec
- Comments: 16,87 sec
- PostHistory: 62 sec
- PostLinks: 0,72 sec
- Posts: 27,38 sec
- Tags: 0,11 sec
- Users: 14,72 sec
- Votes: 23,34 sec

import tables MongoDB doesn't measure time

## Schema information

### *Posts*

You find in `Posts` all non-deleted posts. `PostsWithDeleted` includes rows with deleted posts while sharing the same columns with `Posts` but for deleted posts only a few fields populated which are marked with a [1] below.

- `Id`[1]
- `PostTypeId`[1] *(listed in the `PostTypes` table)*

  1 = Question

  2 = Answer

  3 = Orphaned tag wiki

  4 = Tag wiki excerpt

  5 = Tag wiki

  6 = Moderator nomination

  7 = "Wiki placeholder" *(Appears to include auxiliary site content like the help center introduction, election description, and the tour page's introduction, ask, and don't ask sections)*

  8 = Privilege wiki
- `AcceptedAnswerId` *(only present if `PostTypeId = 1`)*
- `ParentId`[1] *(only present if `PostTypeId = 2`)*
- `CreationDate`[1]
- `DeletionDate`[1] *(only non-null for the SEDE `PostsWithDeleted` table. Deleted posts are not present on `Posts`. Column not present on data dump.)*

- `Score`[1] *(generally non-zero for only Questions, Answers, and Moderator Nominations)*
- `ViewCount` *(nullable)*
- `Body` *(as rendered HTML, not Markdown)*
- `OwnerUserId` *(only present if user has not been deleted; always -1 for tag wiki entries, i.e. the community user owns them)*
- `OwnerDisplayName` *(nullable)*
- `LastEditorUserId` *(nullable)*
- `LastEditorDisplayName` *(nullable)*
- `LastEditDate` (e.g. `2009-03-05T22:28:34.823`) - *the date and time of the most recent edit to the post (nullable)*
- `LastActivityDate` (e.g. `2009-03-11T12:51:01.480`) - *datetime of the post's most recent activity*
- `Title` - *question title (`PostTypeId = 1`), or on Stack Overflow, the tag name for some tag wikis and excerpts (`PostTypeId = 4/5`)*
- `Tags`[1] - *question tags (`PostTypeId = 1`), or on Stack Overflow, the subject tag of some tag wikis and excerpts (`PostTypeId = 4/5`)*
- `AnswerCount` - *the number of undeleted answers (only present if `PostTypeId = 1`)*
- `CommentCount` *(nullable)*
- `FavoriteCount` *(nullable)*
- `ClosedDate`[1] *(present only if the post is closed)*
- `CommunityOwnedDate` *(present only if post is community wiki'd)*
- `ContentLicense`[1]

## *Users*

- `Id`
- `Reputation`
- `CreationDate`
- `DisplayName`
- `LastAccessDate` *(Datetime user last loaded a page; updated every 30 min at most)*
- `WebsiteUrl`
- `Location`
- `AboutMe`
- `Views` *(Number of times the profile is viewed)*
- `UpVotes` *(How many upvotes the user has cast)*
- `DownVotes`
- `ProfileImageUrl`

- `EmailHash` *(now always blank)*
- `AccountId` *(User's Stack Exchange Network profile ID)*

## *Comments*

- `Id`
- `PostId`
- `Score`
- `Text` *(Comment body)*
- `CreationDate`
- `UserDisplayName`
- `UserId` *(Optional. Absent if user has been deleted)*
- `ContentLicense`

## *Badges*

- `Id`
- `UserId`
- `Name` *(Name of the badge)*
- `Date` (e.g. `2008-09-15T08:55:03.923`)
- `Class`

  1 = Gold

  2 = Silver

  3 = Bronze
- `TagBased` = *`True` if badge is for a tag, otherwise it is a named badge*

## *PostHistory*

*(Note that the history of deleted posts is scrubbed from this table in SEDE.)*

- `Id`
- `PostHistoryTypeId` *(listed in the `PostHistoryTypes` table)*

  1 = Initial Title - initial title *(questions only)*

  2 = Initial Body - initial post raw body text

  3 = Initial Tags - initial list of tags *(questions only)*

  4 = Edit Title - modified title *(questions only)*

  5 = Edit Body - modified post body (raw markdown)

  6 = Edit Tags - modified list of tags *(questions only)*

  7 = Rollback Title - reverted title *(questions only)*

8 = Rollback Body - reverted body (raw markdown)

9 = Rollback Tags - reverted list of tags *(questions only)*

10 = Post Closed - post voted to be closed

11 = Post Reopened - post voted to be reopened

12 = Post Deleted - post voted to be removed

13 = Post Undeleted - post voted to be restored

14 = Post Locked - post locked by moderator

15 = Post Unlocked - post unlocked by moderator

16 = Community Owned - post now community owned

17 = Post Migrated - post migrated - *now replaced by 35/36 (away/here)*

18 = Question Merged - question merged with deleted question

19 = Question Protected - question was protected by a moderator.

20 = Question Unprotected - question was unprotected by a moderator.

21 = Post Disassociated - OwnerUserId removed from post by admin

22 = Question Unmerged - answers/votes restored to previously merged question

24 = Suggested Edit Applied

25 = Post Tweeted

31 = Comment discussion moved to chat

33 = Post notice added - *`comment` contains foreign key to PostNotices*

34 = Post notice removed - *`comment` contains foreign key to PostNotices*

35 = Post migrated away - *replaces id 17*

36 = Post migrated here - *replaces id 17*

37 = Post merge source

38 = Post merge destination

50 = Bumped by Community User

52 = Question became hot network question (main) / Hot Meta question (meta)

53 = Question removed from hot network/meta questions by a moderator

66 = Created from Ask Wizard

### Additionally, in older dumps *(all guesses, all seem no longer present in the wild):*

23 = Unknown dev related event

26 = Vote nullification by dev *(ERM?)*

27 = Post unmigrated/hidden moderator migration?

28 = Unknown suggestion event

29 = Unknown moderator event *(possibly de-wikification?)*

30 = Unknown event *(too rare to guess)*

- PostId
- RevisionGUID: At times more than one type of history record can be recorded by a single action. All of these will be grouped using the same RevisionGUID
- CreationDate (e.g. 2009-03-05T22:28:34.823)
- UserId
- UserDisplayName: populated if a user has been removed and no longer referenced by user Id
- Comment: This field will contain the comment made by the user who edited a post.
    - If PostHistoryTypeId = 10, this field contains the CloseReasonId of the close reason (listed in CloseReasonTypes):

      ***Old close reasons:***

      1 = Exact Duplicate

      2 = Off-topic

      3 = Subjective and argumentative

      4 = Not a real question

      7 = Too localized

      10 = General reference

      20 = Noise or pointless (Meta sites only)

      ***Current close reasons:***

      101 = Duplicate

      102 = Off-topic

      103 = Unclear what you're asking

      104 = Too broad

      105 = Primarily opinion-based
    - If PostHistoryTypeId in (33,34) this field contains the PostNoticeId of the PostNotice
- Text: A raw version of the new value for a given revision

  - If PostHistoryTypeId in (10,11,12,13,14,15,19,20,35) this column will contain a

  JSON encoded string with all users who have voted for the PostHistoryTypeId

  - If it is a duplicate close vote, the JSON string will contain an array of original questions as

  OriginalQuestionIds

- If `PostHistoryTypeId = 17` this column will contain migration details of either `from <url>` or `to <url>`

- `ContentLicense`

## *PostLinks*

- `Id` primary key
- `CreationDate` when the link was created
- `PostId` id of source post
- `RelatedPostId` id of target/related post
- `LinkTypeId` type of link

1 = Linked (`PostId` contains a link to `RelatedPostId`)

3 = Duplicate (`PostId` is a duplicate of `RelatedPostId`)

## *Tags*

- `Id`
- `TagName`
- `Count`
- `ExcerptPostId` *(nullable) Id of Post that holds the excerpt text of the tag*
- `WikiPostId` *(nullable) Id of Post that holds the wiki text of the tag*
- `IsModeratorOnly`
- `IsRequired`

## *Votes*

- `Id`
- `PostId`
- `VoteTypeId` *(listed in the* `VoteTypes` *table)*

1 = AcceptedByOriginator

2 = UpMod *(AKA upvote)*

3 = DownMod *(AKA downvote)*

4 = Offensive

5 = Favorite *(AKA bookmark; `UserId` will also be populated) feature removed after October 2022 / replaced by Saves*

6 = Close (effective 2013-06-25: Close votes are **only** stored in table: `PostHistory`)

7 = Reopen

8 = BountyStart *(`UserId` and `BountyAmount` will also be populated)*

9 = BountyClose *(`BountyAmount` will also be populated)*

10 = Deletion

11 = Undeletion

12 = Spam

15 = ModeratorReview *(i.e., a moderator looking at a flagged post)*

16 = ApproveEditSuggestion

- `UserId` *(present only if `VoteTypeId in (5,8)`; -1 if user is deleted)*
- `CreationDate` Date only (`2018-07-31 00:00:00` *time data is purposefully removed to protect user privacy)*
- `BountyAmount` (present only if `VoteTypeId in (8,9)`)

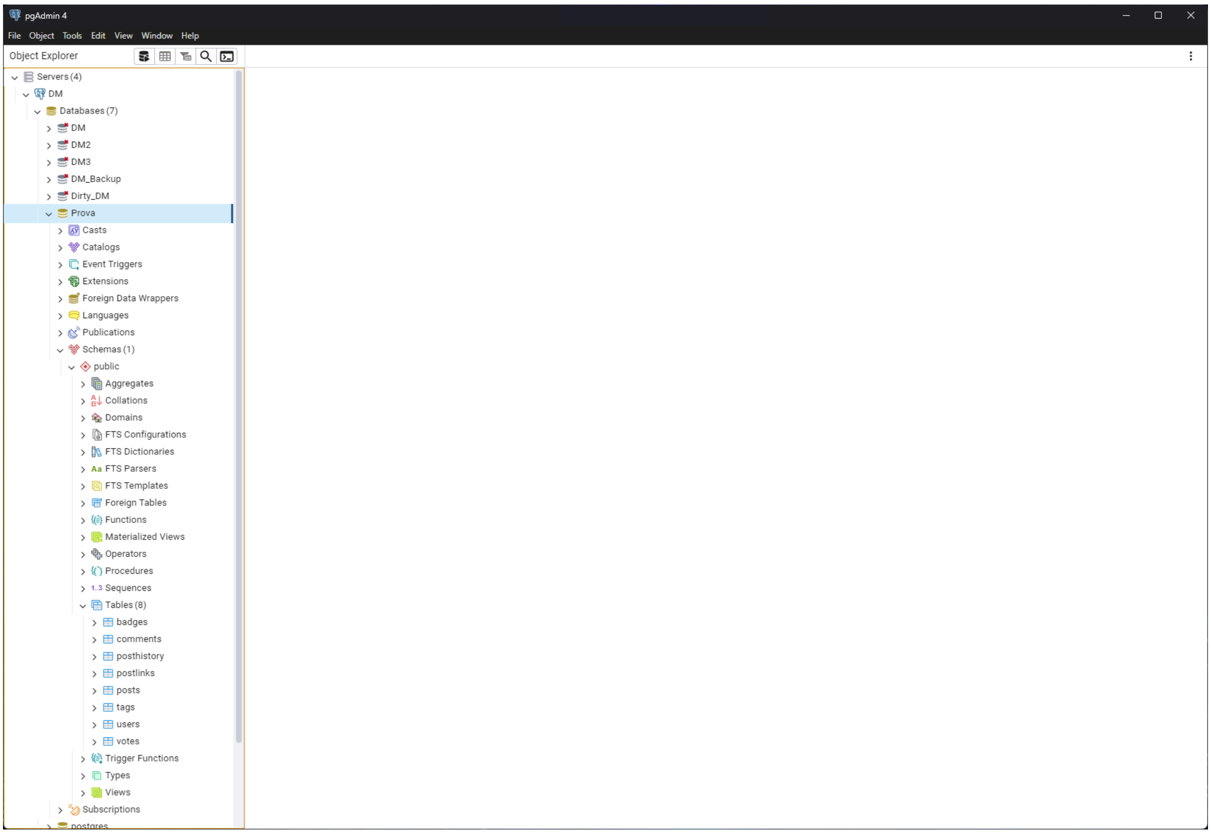
# Comparison of the interfaces

**A. Visual Design & Layout**

• **pgAdmin:**

• Describe the overall layout (e.g., panel arrangements, the use of menus and dashboards).

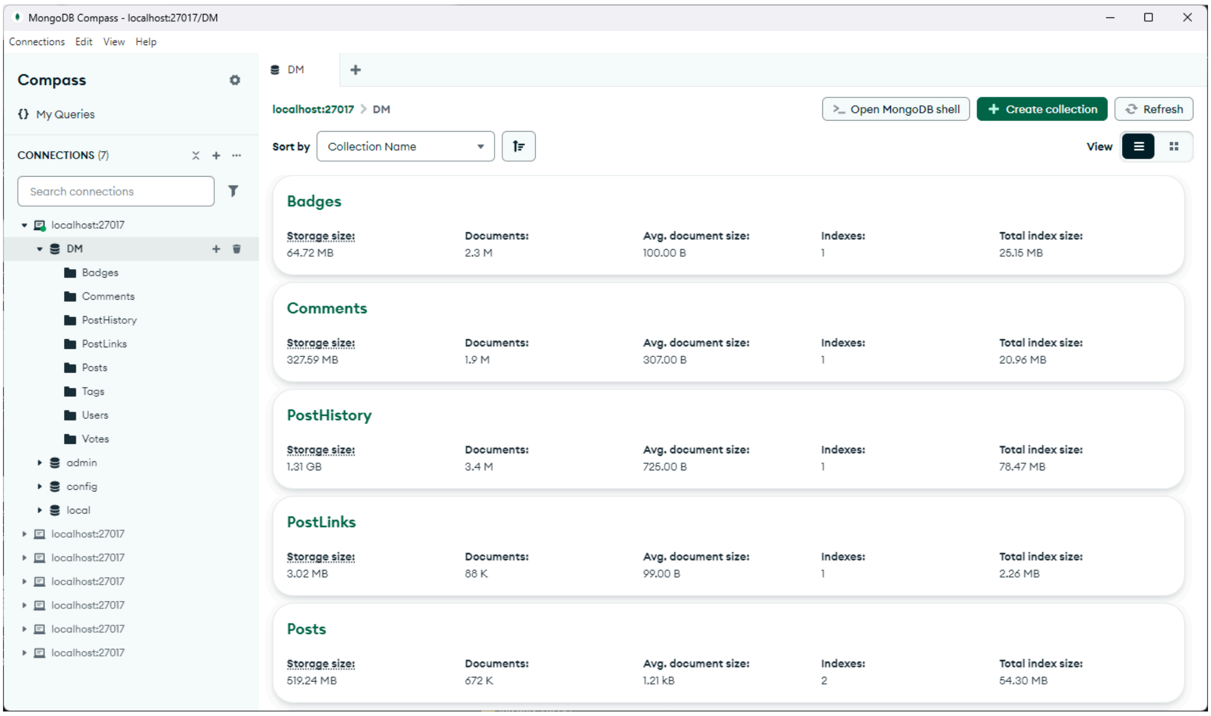
• Comment on the consistency of visual elements and how the design supports a structured, relational model.



- **MongoDB Compass:**

- Discuss how Compass organizes data (e.g., schema visualization, document trees).

- Consider whether the layout supports exploratory data analysis and flexible data handling.



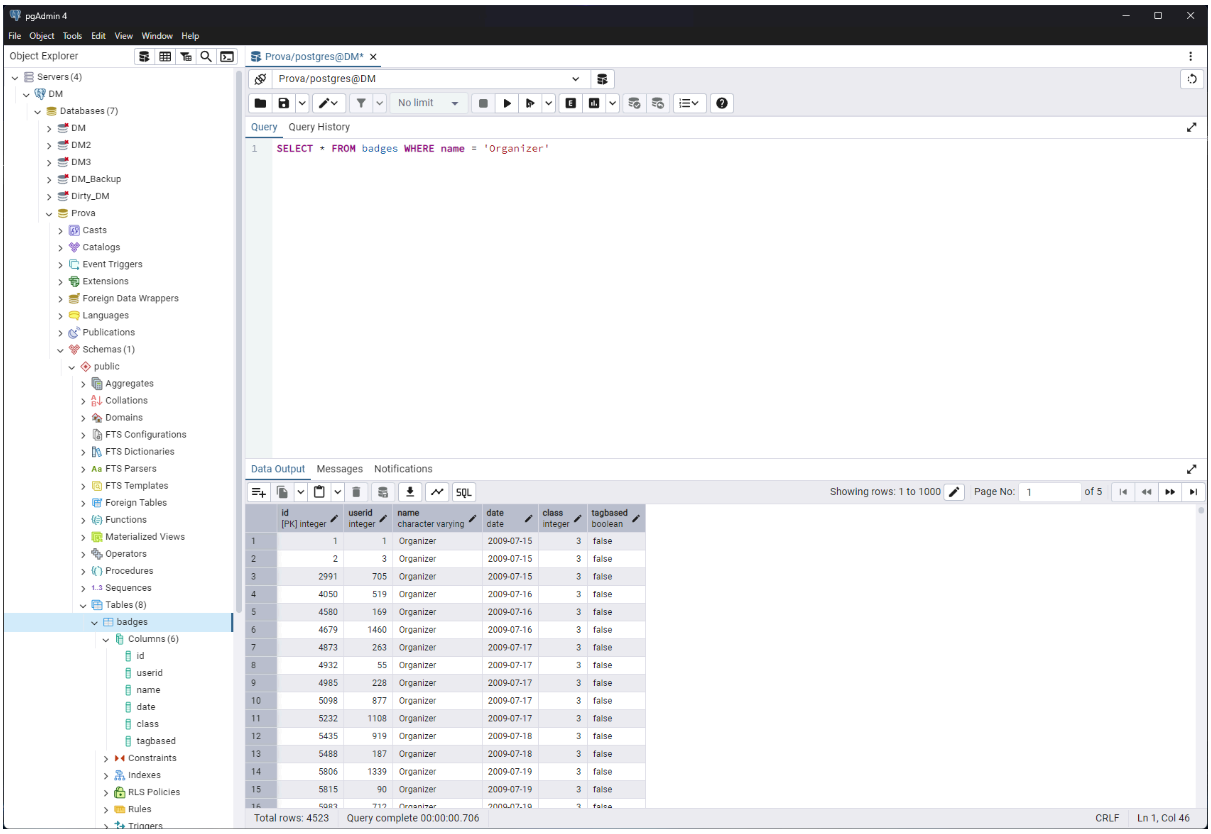
| | Storage size: | Documents: | Avg. document size: | Indexes: | Total index size: |
|---|---|---|---|---|---|
| Badges | 64.72 MB | 2.3 M | 100.00 B | 1 | 25.15 MB |
| Comments | 327.59 MB | 1.9 M | 307.00 B | 1 | 20.96 MB |
| PostHistory | 1.31 GB | 3.4 M | 725.00 B | 1 | 78.47 MB |
| PostLinks | 3.02 MB | 88 K | 99.00 B | 1 | 2.26 MB |
| Posts | 519.24 MB | 672 K | 1.21 kB | 2 | 54.30 MB |

## B. Navigation & Usability

### • Menu Structure & Workflow:

• Compare how each tool organizes access to key functions such as running queries, managing schemas/collections, or monitoring performance.

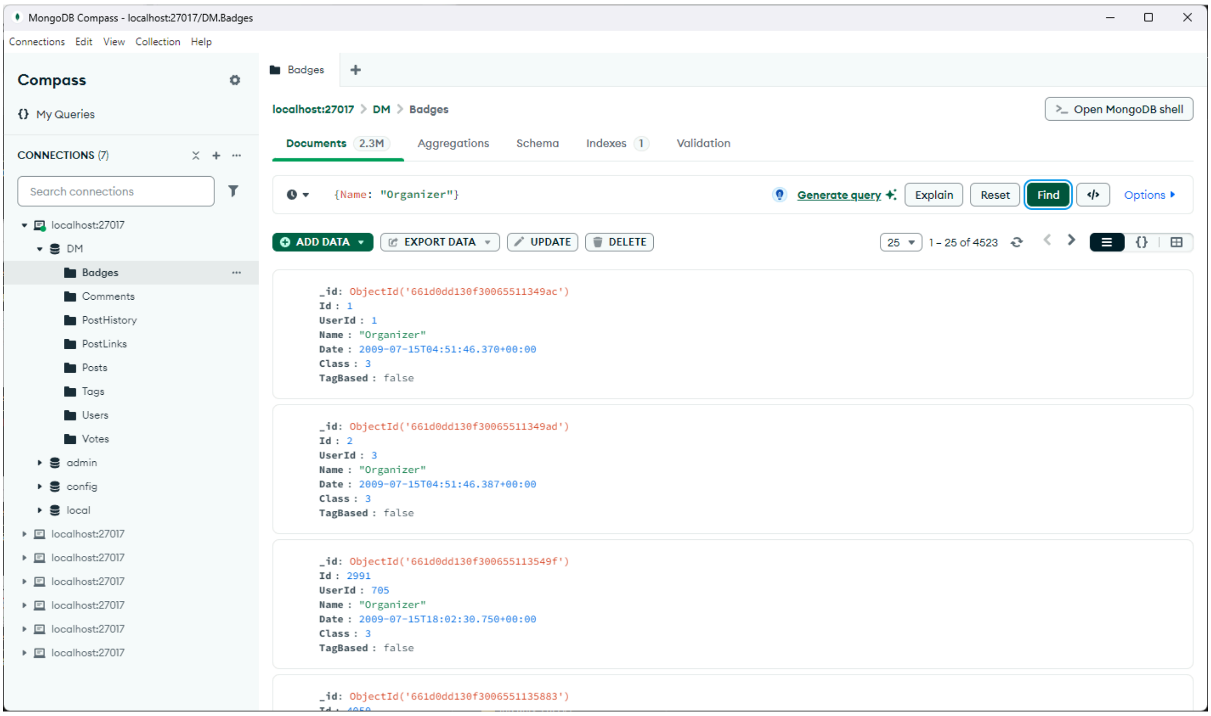
• Evaluate the intuitiveness of their navigation and the ease with which a user can complete common tasks.



```
1  SELECT * FROM badges WHERE name = 'Organizer'
```

| | id<br>[PK] integer | userid<br>integer | name<br>character varying | date<br>date | class<br>integer | tagbased<br>boolean |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Organizer | 2009-07-15 | 3 | false |
| 2 | 2 | 3 | Organizer | 2009-07-15 | 3 | false |
| 3 | 2991 | 705 | Organizer | 2009-07-15 | 3 | false |
| 4 | 4050 | 519 | Organizer | 2009-07-16 | 3 | false |
| 5 | 4580 | 169 | Organizer | 2009-07-16 | 3 | false |
| 6 | 4679 | 1460 | Organizer | 2009-07-16 | 3 | false |
| 7 | 4873 | 263 | Organizer | 2009-07-17 | 3 | false |
| 8 | 4932 | 55 | Organizer | 2009-07-17 | 3 | false |
| 9 | 4985 | 228 | Organizer | 2009-07-17 | 3 | false |
| 10 | 5098 | 877 | Organizer | 2009-07-17 | 3 | false |
| 11 | 5232 | 1108 | Organizer | 2009-07-17 | 3 | false |
| 12 | 5435 | 919 | Organizer | 2009-07-18 | 3 | false |
| 13 | 5488 | 187 | Organizer | 2009-07-18 | 3 | false |
| 14 | 5806 | 1339 | Organizer | 2009-07-19 | 3 | false |
| 15 | 5815 | 90 | Organizer | 2009-07-19 | 3 | false |
| 16 | 5983 | 712 | Organizer | 2009-07-19 | 3 | false |

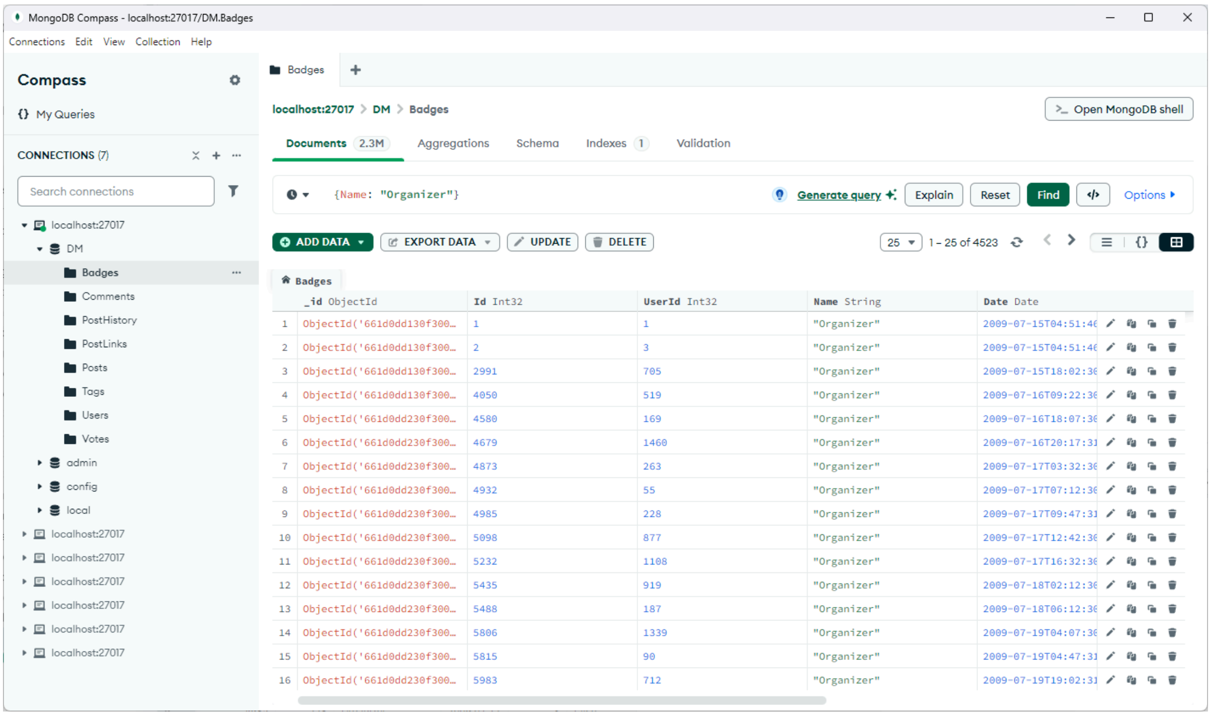
Total rows: 4523 Query complete 00:00:00.706



```
{Name: "Organizer"}
```

```
_id: ObjectId('661d0dd130f30065511349ac')
Id : 1
UserId : 1
Name : "Organizer"
Date : 2009-07-15T04:51:46.370+00:00
Class : 3
TagBased : false

_id: ObjectId('661d0dd130f30065511349ad')
Id : 2
UserId : 3
Name : "Organizer"
Date : 2009-07-15T04:51:46.387+00:00
Class : 3
TagBased : false

_id: ObjectId('661d0dd130f300655113549f')
Id : 2991
UserId : 705
Name : "Organizer"
Date : 2009-07-15T18:02:30.750+00:00
Class : 3
TagBased : false

_id: ObjectId('661d0dd130f3006551135883')
```

MongoDB Compass allows to view the documents in a table view too



```
{Name: "Organizer"}
```

Badges

| | _id ObjectId | Id Int32 | UserId Int32 | Name String | Date Date |
|---|---|---|---|---|---|
| 1 | ObjectId('661d0dd130f300… | 1 | 1 | "Organizer" | 2009-07-15T04:51:46 |
| 2 | ObjectId('661d0dd130f300… | 2 | 3 | "Organizer" | 2009-07-15T04:51:46 |
| 3 | ObjectId('661d0dd130f300… | 2991 | 705 | "Organizer" | 2009-07-15T18:02:30 |
| 4 | ObjectId('661d0dd130f300… | 4050 | 519 | "Organizer" | 2009-07-16T09:22:30 |
| 5 | ObjectId('661d0dd230f300… | 4580 | 169 | "Organizer" | 2009-07-16T18:07:30 |
| 6 | ObjectId('661d0dd230f300… | 4679 | 1460 | "Organizer" | 2009-07-16T20:17:31 |
| 7 | ObjectId('661d0dd230f300… | 4873 | 263 | "Organizer" | 2009-07-17T03:32:30 |
| 8 | ObjectId('661d0dd230f300… | 4932 | 55 | "Organizer" | 2009-07-17T07:12:30 |
| 9 | ObjectId('661d0dd230f300… | 4985 | 228 | "Organizer" | 2009-07-17T09:47:31 |
| 10 | ObjectId('661d0dd230f300… | 5098 | 877 | "Organizer" | 2009-07-17T12:42:30 |
| 11 | ObjectId('661d0dd230f300… | 5232 | 1108 | "Organizer" | 2009-07-17T16:32:30 |
| 12 | ObjectId('661d0dd230f300… | 5435 | 919 | "Organizer" | 2009-07-18T02:12:30 |
| 13 | ObjectId('661d0dd230f300… | 5488 | 187 | "Organizer" | 2009-07-18T06:12:30 |
| 14 | ObjectId('661d0dd230f300… | 5806 | 1339 | "Organizer" | 2009-07-19T04:07:30 |
| 15 | ObjectId('661d0dd230f300… | 5815 | 90 | "Organizer" | 2009-07-19T04:47:31 |
| 16 | ObjectId('661d0dd230f300… | 5983 | 712 | "Organizer" | 2009-07-19T19:02:31 |

## C. Feature Set & Functionality

### • Core Tools:

• For pgAdmin, discuss features like query builders, SQL editors, and support for managing relational schemas.

- For MongoDB Compass, focus on features such as JSON document visualization, aggregation pipelines, and real-time performance metrics.

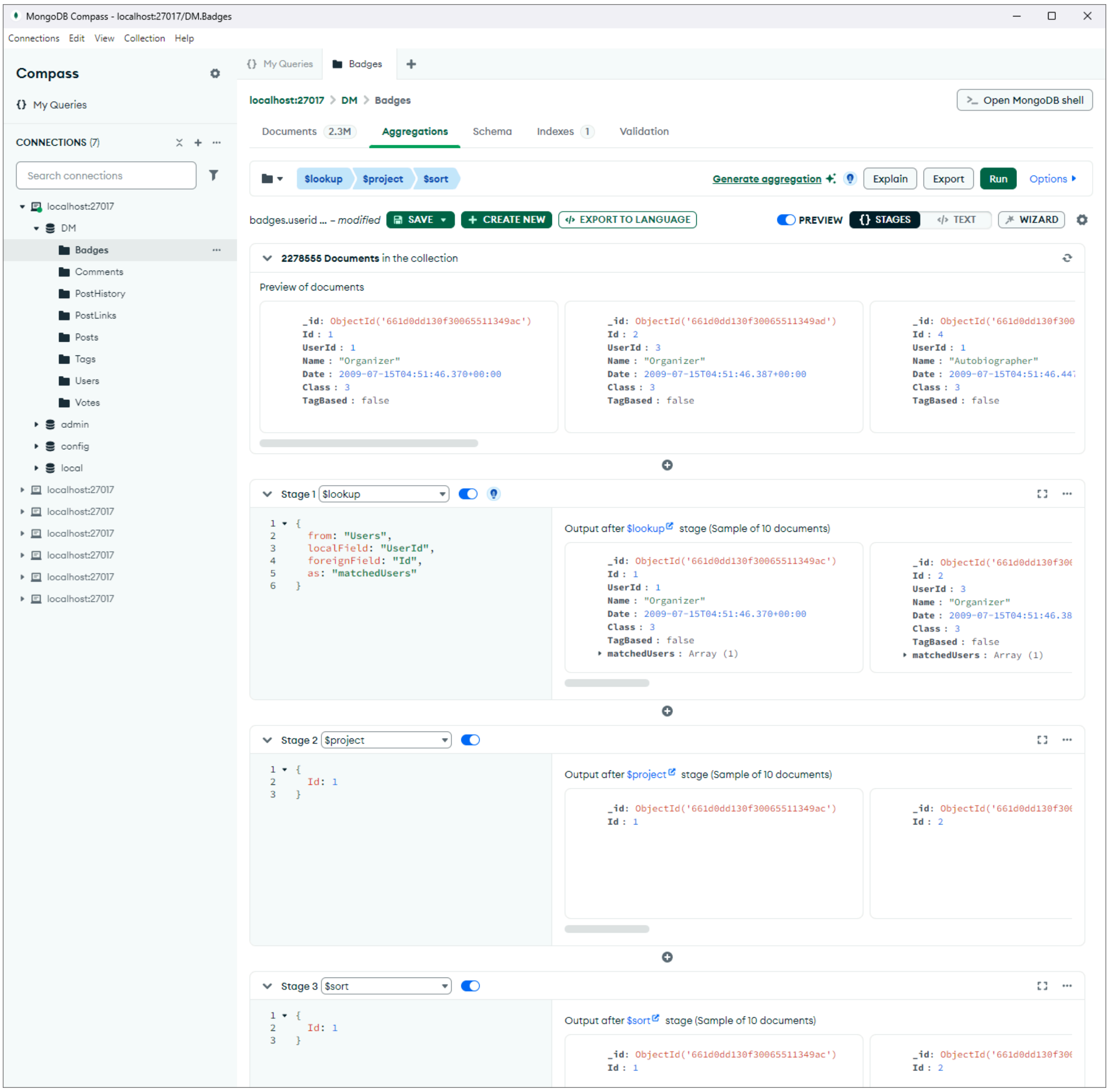
- **Advanced Capabilities:**

- Highlight any additional tools (e.g., data export/import, index management, or monitoring dashboards) and discuss their depth and ease of use.

The query tool provided by pgAdmin is as simple as they come, a text editor where to write your queries.

There is a set of predetermined queries available when right clicking on a table. (viewing the data, CRUD scripts.

MongoDB Compass provides a useful interface to compose aggregation pipelines.



badges.userid ... – *modified*

**2278555 Documents** in the collection

Preview of documents

```
_id: ObjectId('661d0dd130f30065511349ac')
Id : 1
UserId : 1
Name : "Organizer"
Date : 2009-07-15T04:51:46.370+00:00
Class : 3
TagBased : false
```

```
_id: ObjectId('661d0dd130f30065511349ad')
Id : 2
UserId : 3
Name : "Organizer"
Date : 2009-07-15T04:51:46.387+00:00
Class : 3
TagBased : false
```

```
_id: ObjectId('661d0dd130f300
Id : 4
UserId : 1
Name : "Autobiographer"
Date : 2009-07-15T04:51:46.447
Class : 3
TagBased : false
```

Stage 1 $lookup

```
{
   from: "Users",
   localField: "UserId",
   foreignField: "Id",
   as: "matchedUsers"
}
```

Output after $lookup stage (Sample of 10 documents)

```
_id: ObjectId('661d0dd130f30065511349ac')
Id : 1
UserId : 1
Name : "Organizer"
Date : 2009-07-15T04:51:46.370+00:00
Class : 3
TagBased : false
matchedUsers : Array (1)
```

```
_id: ObjectId('661d0dd130f30(
Id : 2
UserId : 3
Name : "Organizer"
Date : 2009-07-15T04:51:46.38
Class : 3
TagBased : false
matchedUsers : Array (1)
```

Stage 2 $project

```
{
   Id: 1
}
```

Output after $project stage (Sample of 10 documents)

```
_id: ObjectId('661d0dd130f30065511349ac')
Id : 1
```

```
_id: ObjectId('661d0dd130f30(
Id : 2
```

Stage 3 $sort

```
{
   Id: 1
}
```

Output after $sort stage (Sample of 10 documents)

```
_id: ObjectId('661d0dd130f30065511349ac')
Id : 1
```

```
_id: ObjectId('661d0dd130f30(
Id : 2
```

both of the GUI softwares allow to import/export data as well as a CLI window, if the user prefers to use that instead.

# Comparison of the query language

## Overview

### MongoDB Query Language

• **Document-Based Syntax:** MongoDB uses a JSON/BSON syntax to construct queries. Operations are typically issued as methods on a collection object. For example, to find users over 30:

```
db[“Users”].find({ age: { $gt: 30 } })
```

• **Operators and Expressions:** MQL supports a rich set of operators (e.g., $eq, $gt, $in, $and, $or) that allow filtering on nested or even array fields. This fits well with MongoDB’s schema-less, nested documents.

• **Aggregation Framework:** Instead of traditional SQL GROUP BY, MongoDB provides a pipeline of stages (such as $match, $group, $project, $sort) that allow you to transform and compute over data in a step-by-step fashion.

• **Joins via $lookup:** Although MongoDB does not support traditional joins (because denormalization is common), it has introduced the $lookup operator in the aggregation pipeline to combine documents from different collections when needed.

• **Flexibility:** Because MongoDB is designed to work with unstructured or semi-structured data, its query language is highly dynamic. You can add new fields or change the structure of documents without having to modify a rigid schema.

### PostgreSQL Query Language

• **Declarative, ANSI-Compliant Syntax:** PostgreSQL relies on SQL—a well-established, standardized language. A typical query to retrieve users over 30 would be:

```
SELECT * FROM users WHERE age > 30;
```

• **Structured and Schema-Driven:** SQL is designed for data stored in clearly defined tables. Columns, data types, and relationships are specified in advance, which enables the use of strong data integrity constraints.

• **Powerful Joins and Subqueries:** PostgreSQL supports various types of joins (INNER, LEFT, RIGHT, FULL) and subqueries. Complex queries can easily combine data from multiple tables, enforce referential integrity, and compute aggregates.

• **Advanced Aggregation and Window Functions:** Aggregation is performed with GROUP BY clauses and supported by an extensive set of aggregate functions. PostgreSQL also supports window functions and common table expressions (WITH queries) to handle complex data analysis.

- **Extensibility:** PostgreSQL’s SQL is not only standard but can be extended through user-defined functions, stored procedures, and support for several procedural languages, which can encapsulate business logic close to the data.

## Key Differences

### Syntax and Structure

**MongoDB**:

- Uses JSON-like documents to specify query criteria.

- Queries are written as object literals, which can be more natural for developers working in JavaScript or other object-oriented languages.

```
db[“Orders”].find({ status: "shipped", total: { $gte: 50 } })
```

**PostgreSQL:**

- Uses a declarative, text-based language with clear keywords.

- The language is standardized, making it easier to port skills between different SQL databases.

```
SELECT * FROM orders WHERE status = 'shipped' AND total >= 50;
```

### Data Model

**MongoDB:**

- Designed for hierarchical, nested data. Its query language naturally expresses conditions on embedded documents and arrays.

- Aggregation pipelines can perform multi-stage transformations that align with the flexible document model.

**PostgreSQL:**

- Optimized for flat, tabular data. SQL queries express relationships and joins between tables through well-defined keys.

- Complex aggregations, subqueries, and window functions enable robust data analysis on structured datasets.

### Joins and Relationship Handling

**MongoDB:**

- Lacks a native, SQL-like join operation because the data is often stored in denormalized documents.

- Uses the $lookup stage in aggregation pipelines for joining documents from different collections—but with more limited capabilities compared to SQL joins.

**PostgreSQL:**

- Joins are first-class citizens. You can easily join tables with various join types, which is essential for normalized data models.

- The declarative nature of SQL joins makes it straightforward to enforce relationships and retrieve combined datasets.

### Aggregation and Data Transformation

**MongoDB:**

- The aggregation pipeline provides a procedural approach where each stage transforms the data gradually.

- This can be very powerful for operations such as grouping, filtering, and reshaping data within a document-centric model.

**PostgreSQL:**

- Uses aggregate functions with GROUP BY clauses and advanced window functions to compute summaries.

- The set-based nature of SQL makes it efficient for operations that work across rows of data in a table.

## Performance

- **Query Optimizers:**

PostgreSQL’s query planner and optimizer work on the declarative SQL, often leading to highly optimized execution plans (especially for complex joins and aggregates).

MongoDB’s optimizer is tuned for its document model and can quickly return results when data is denormalized and indexed appropriately.

- **Indexing:**

Both databases offer robust indexing options. PostgreSQL indexes work on table columns, while MongoDB indexes can target nested fields within documents.

• **Flexibility vs. Consistency:**

MongoDB's MQL provides rapid development and flexibility—ideal for applications where the schema evolves over time. PostgreSQL's SQL, meanwhile, favors consistency, data integrity, and complex transactional support.

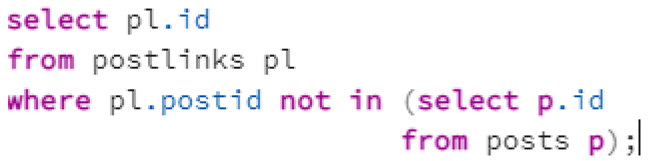
# Integrity checks on the dataset

Before starting to compare the performance of the different systems, we should carry out some integrity checks, necessary for the RDBMS to work properly.

## - every postlinks.postid should be contained in posts.id

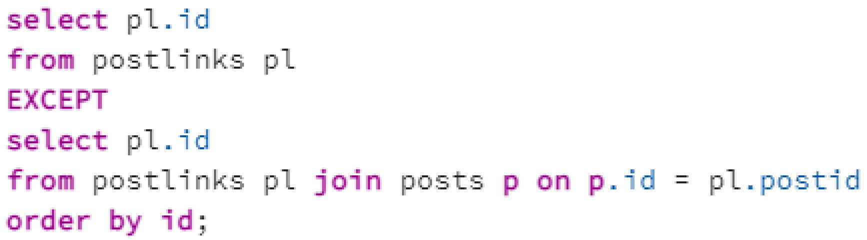
### In PostgreSQL

The first integrity check that is performed is: postlinks.postid should be contained in posts.id
To check if there are any postid inside the postlinks relation, we can perform this query:



```
select pl.id
from postlinks pl
where pl.postid not in (select p.id
                        from posts p);
```

This turns out to be very inefficient, completing in **2 hours and 37 minutes**.
PostgreSQL is more efficient when performing joins, which are very frequent in relational datasets, in fact when performing the equivalent query:

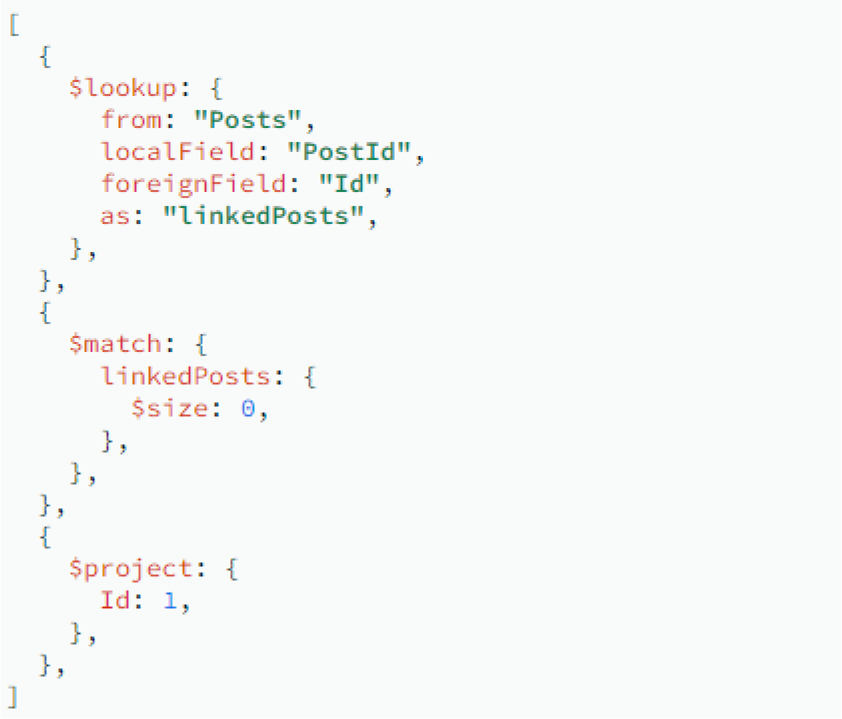


```
select pl.id
from postlinks pl
EXCEPT
select pl.id
from postlinks pl join posts p on p.id = pl.postid
order by id;
```

the same answer is returned after **258 milliseconds**.

### In MongoDB

Performing the same check using the following query in MongoDB

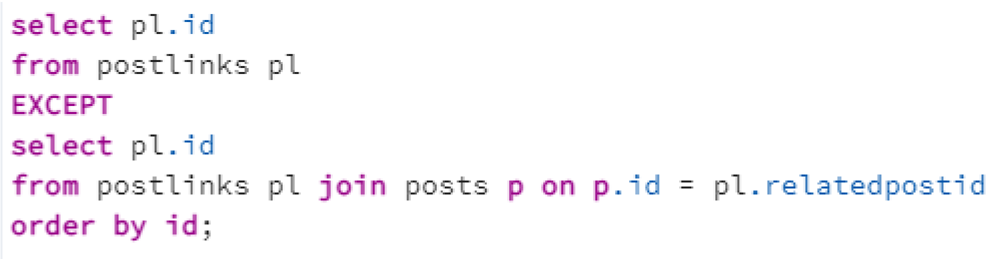


```
[
  {
    $lookup: {
      from: "Posts",
      localField: "PostId",
      foreignField: "Id",
      as: "linkedPosts",
    },
  },
  {
    $match: {
      linkedPosts: {
        $size: 0,
      },
    },
  },
  {
    $project: {
      Id: 1,
    },
  },
]
```

The application returns an error complaining about exceeding the time limit.
To improve performance an index must be created on the Id of the Posts collection.
After the index is created, the above query runs in **1.825 seconds.**

- every postlinks.relatedpostid should be contained in posts.id

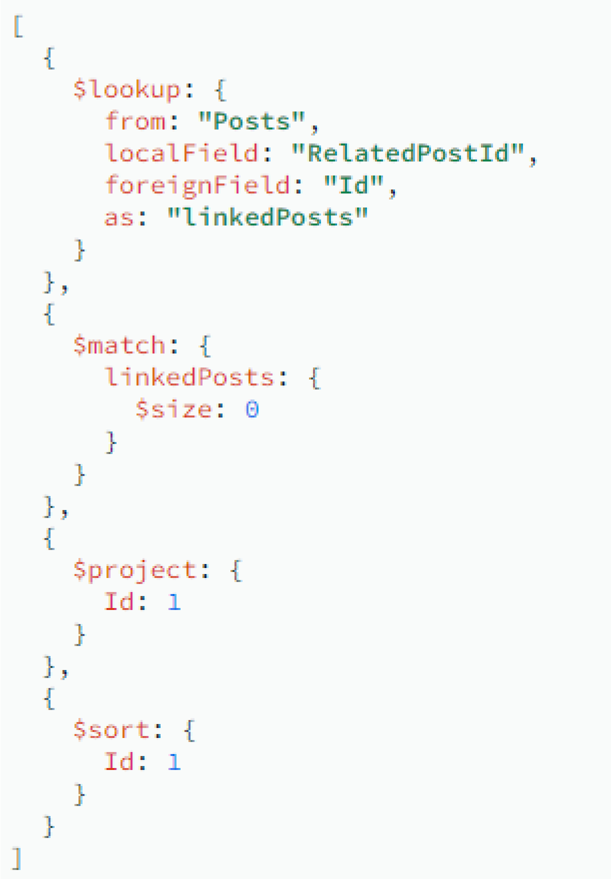
### In PostgreSQL



```
select pl.id
from postlinks pl
EXCEPT
select pl.id
from postlinks pl join posts p on p.id = pl.relatedpostid
order by id;
```

Completed in 302 ms

In MongoDB

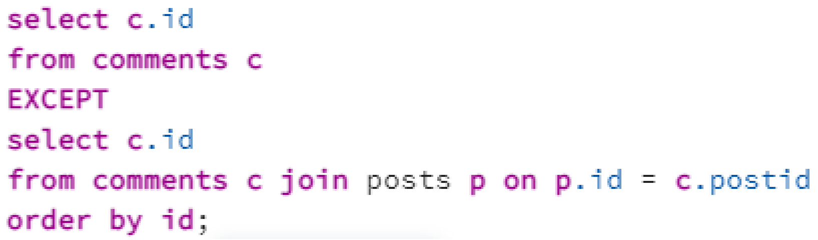


```
[
  {
    $lookup: {
      from: "Posts",
      localField: "RelatedPostId",
      foreignField: "Id",
      as: "linkedPosts"
    }
  },
  {
    $match: {
      linkedPosts: {
        $size: 0
      }
    }
  },
  {
    $project: {
      Id: 1
    }
  },
  {
    $sort: {
      Id: 1
    }
  }
]
```

Completed in 1.920 seconds

- every comments.postid should be contained in posts.id

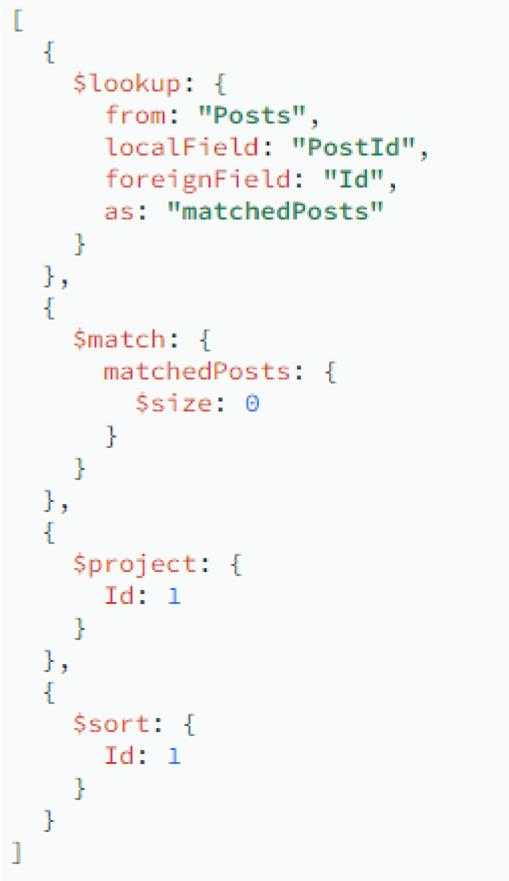
In PostgreSQL



```
select c.id
from comments c
EXCEPT
select c.id
from comments c join posts p on p.id = c.postid
order by id;
```

Completed in 3.172 seconds

In MongoDB

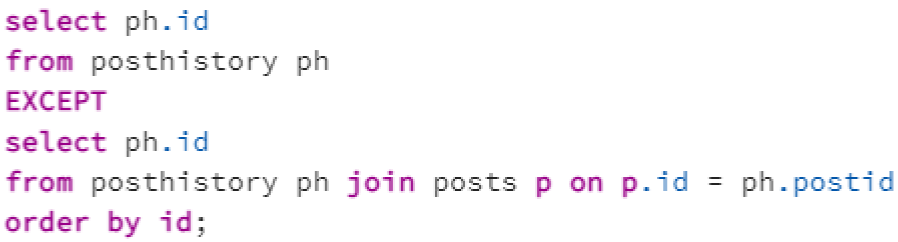


```
[
  {
    $lookup: {
      from: "Posts",
      localField: "PostId",
      foreignField: "Id",
      as: "matchedPosts"
    }
  },
  {
    $match: {
      matchedPosts: {
        $size: 0
      }
    }
  },
  {
    $project: {
      Id: 1
    }
  },
  {
    $sort: {
      Id: 1
    }
  }
]
```

Completed in 27.334 seconds

- every posthistory.postid should be contained in posts.id

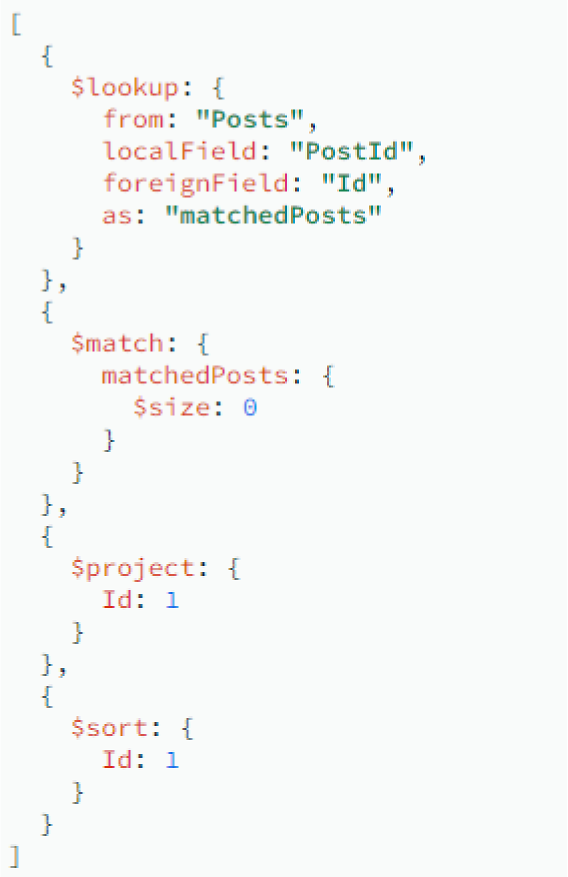
In PostgreSQL



```
select ph.id
from posthistory ph
EXCEPT
select ph.id
from posthistory ph join posts p on p.id = ph.postid
order by id;
```

Completed in 7.4 seconds

In MongoDB

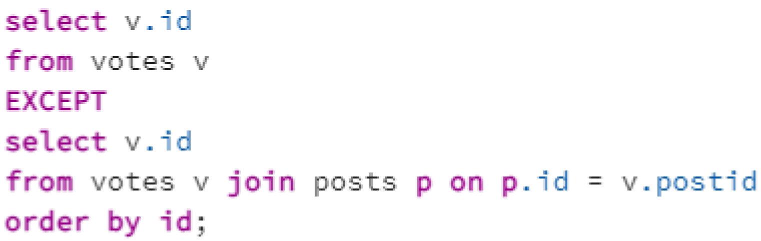


```
[
  {
    $lookup: {
      from: "Posts",
      localField: "PostId",
      foreignField: "Id",
      as: "matchedPosts"
    }
  },
  {
    $match: {
      matchedPosts: {
        $size: 0
      }
    }
  },
  {
    $project: {
      Id: 1
    }
  },
  {
    $sort: {
      Id: 1
    }
  }
]
```

Completed in 51.146 seconds

- every votes.postid should be contained in posts.id

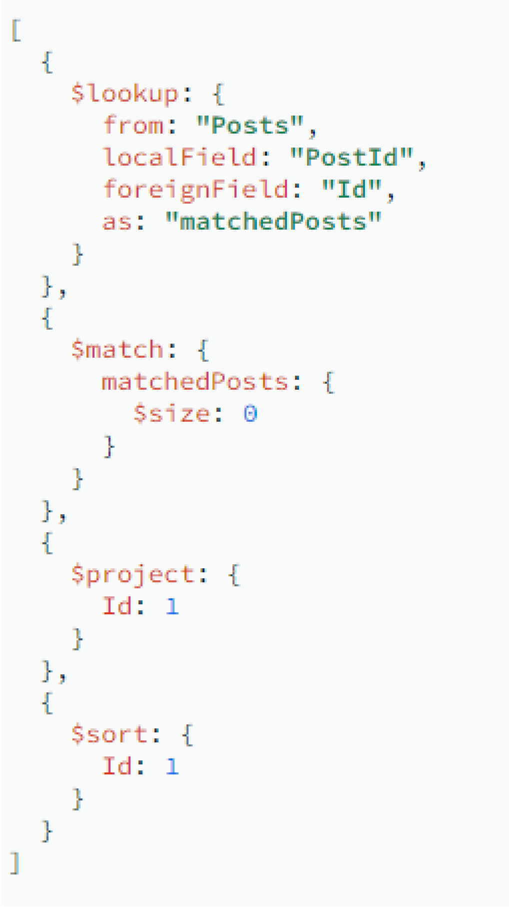
In PostgreSQL



```
select v.id
from votes v
EXCEPT
select v.id
from votes v join posts p on p.id = v.postid
order by id;
```

Completed in 7.914 seconds

In MongoDB

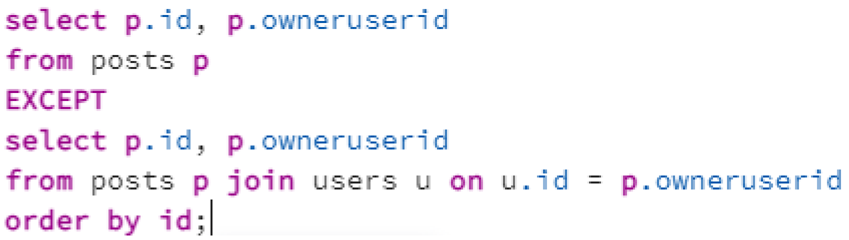


```
[
  {
    $lookup: {
      from: "Posts",
      localField: "PostId",
      foreignField: "Id",
      as: "matchedPosts"
    }
  },
  {
    $match: {
      matchedPosts: {
        $size: 0
      }
    }
  },
  {
    $project: {
      Id: 1
    }
  },
  {
    $sort: {
      Id: 1
    }
  }
]
```

Completed in 76.631 seconds

- every posts.OwnerUserId should be contained in users.id

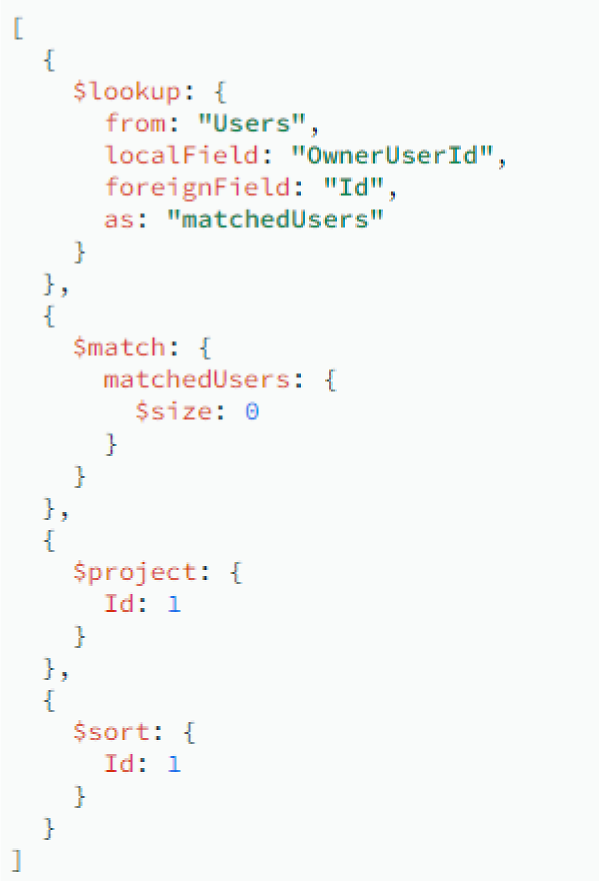
In PostgreSQL



```
select p.id, p.owneruserid
from posts p
EXCEPT
select p.id, p.owneruserid
from posts p join users u on u.id = p.owneruserid
order by id;
```

Completed in 3.838 seconds

In MongoDB

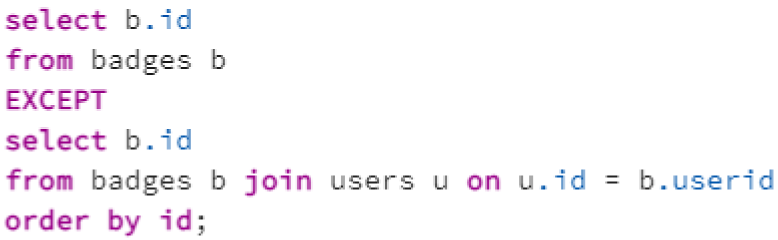


```
[
  {
    $lookup: {
      from: "Users",
      localField: "OwnerUserId",
      foreignField: "Id",
      as: "matchedUsers"
    }
  },
  {
    $match: {
      matchedUsers: {
        $size: 0
      }
    }
  },
  {
    $project: {
      Id: 1
    }
  },
  {
    $sort: {
      Id: 1
    }
  }
]
```

Completed in 18.785 seconds

- every badges.userid should be contained in users.id

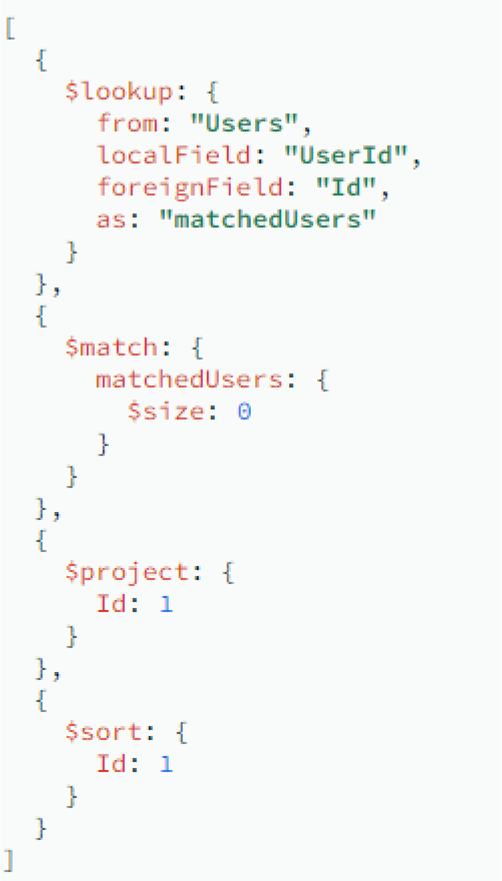
In PostgreSQL



```
select b.id
from badges b
EXCEPT
select b.id
from badges b join users u on u.id = b.userid
order by id;
```

Completed in 3.7 seconds

In MongoDB

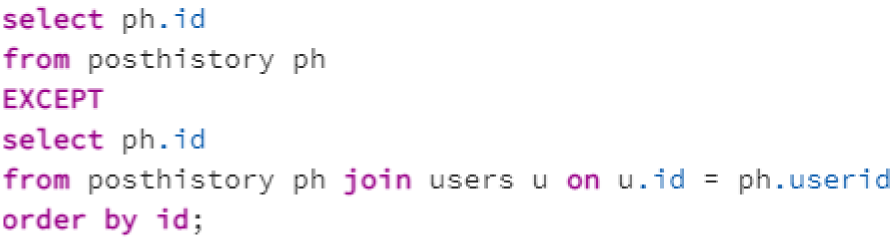


```
[
  {
    $lookup: {
      from: "Users",
      localField: "UserId",
      foreignField: "Id",
      as: "matchedUsers"
    }
  },
  {
    $match: {
      matchedUsers: {
        $size: 0
      }
    }
  },
  {
    $project: {
      Id: 1
    }
  },
  {
    $sort: {
      Id: 1
    }
  }
]
```

Completed in 33.827 seconds

- every posthistory.userid should be contained in in users.id

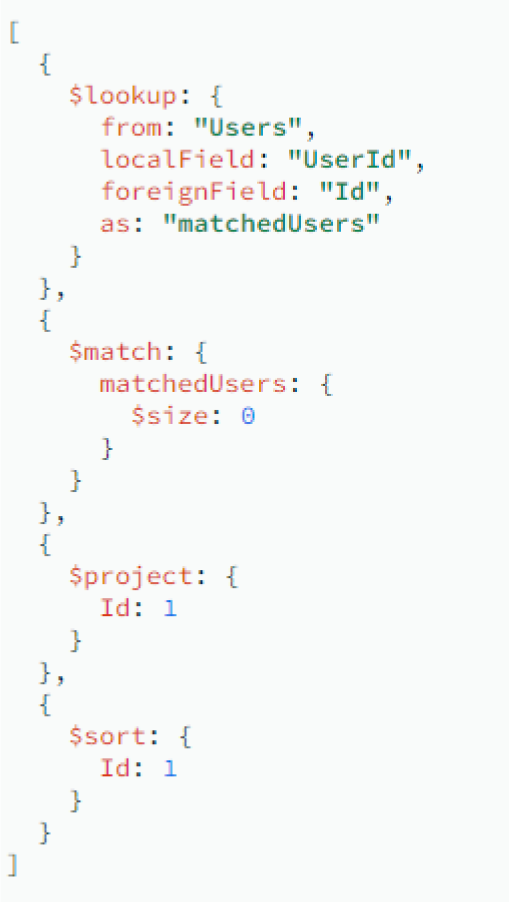
In PostgreSQL



```
select ph.id
from posthistory ph
EXCEPT
select ph.id
from posthistory ph join users u on u.id = ph.userid
order by id;
```

don’t delete they are all null
Completed in 7.112 seconds

### In MongoDB



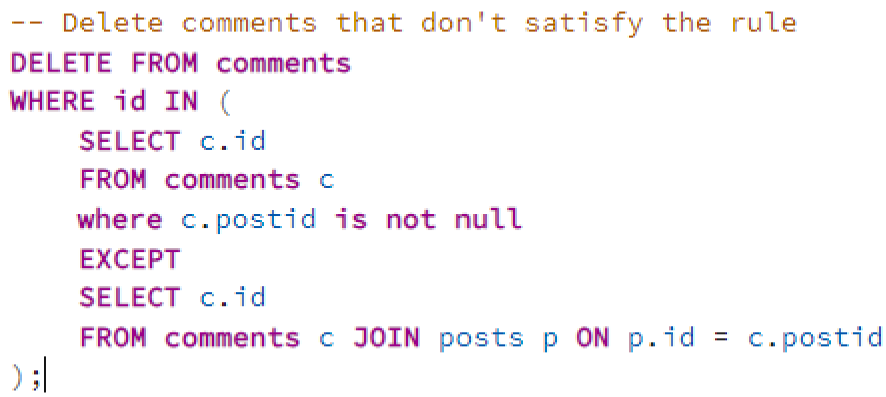
```
[
  {
    $lookup: {
      from: "Users",
      localField: "UserId",
      foreignField: "Id",
      as: "matchedUsers"
    }
  },
  {
    $match: {
      matchedUsers: {
        $size: 0
      }
    }
  },
  {
    $project: {
      Id: 1
    }
  },
  {
    $sort: {
      Id: 1
    }
  }
]
```

Completed in 56.260 seconds

# Cleaning the data

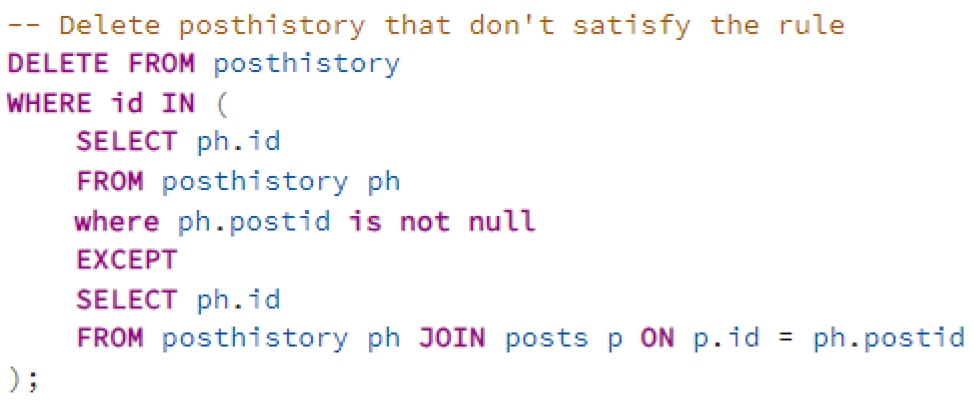
In some cases null values should be allowed, therefore we will not delete those rows, but only the ones that actually break the foreign key with a value.
To be fair to MongoDB, we delete the same data in both databases.

### In PostgreSQL



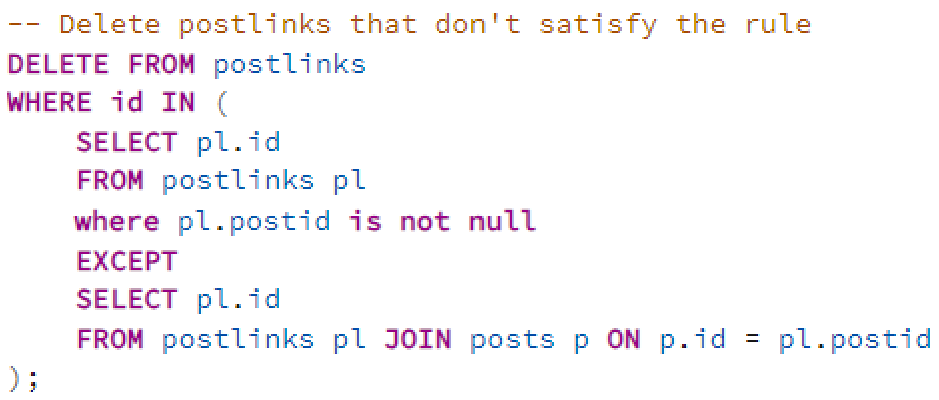
```
-- Delete comments that don't satisfy the rule
DELETE FROM comments
WHERE id IN (
    SELECT c.id
    FROM comments c
    where c.postid is not null
    EXCEPT
    SELECT c.id
    FROM comments c JOIN posts p ON p.id = c.postid
);
```

Completed in 8.762 seconds (deleted 60534)



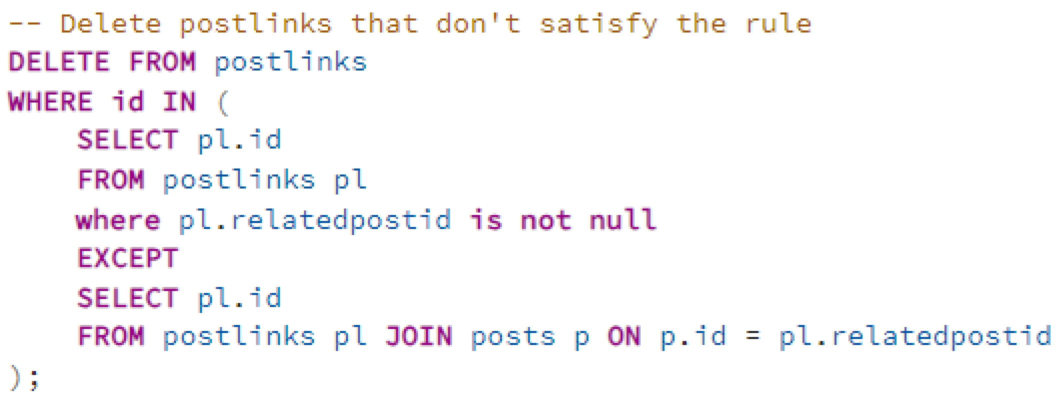
```
-- Delete posthistory that don't satisfy the rule
DELETE FROM posthistory
WHERE id IN (
    SELECT ph.id
    FROM posthistory ph
    where ph.postid is not null
    EXCEPT
    SELECT ph.id
    FROM posthistory ph JOIN posts p ON p.id = ph.postid
);
```

Completed in 29.958 seconds (deleted 197124)



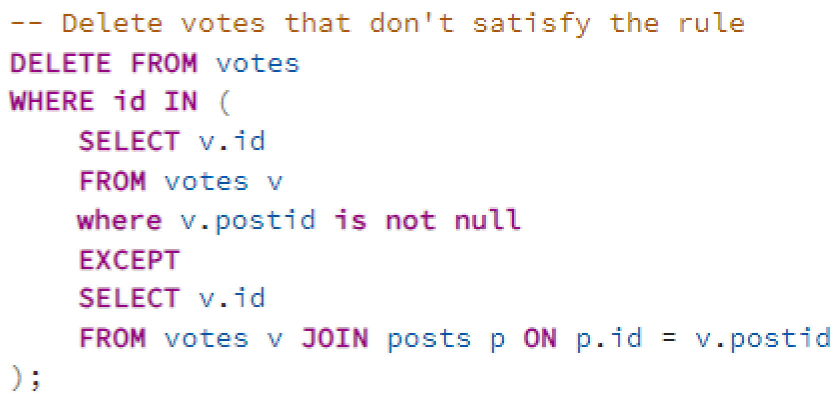
```
-- Delete postlinks that don't satisfy the rule
DELETE FROM postlinks
WHERE id IN (
    SELECT pl.id
    FROM postlinks pl
    where pl.postid is not null
    EXCEPT
    SELECT pl.id
    FROM postlinks pl JOIN posts p ON p.id = pl.postid
);
```

Completed in 3.978 (deleted 7055)



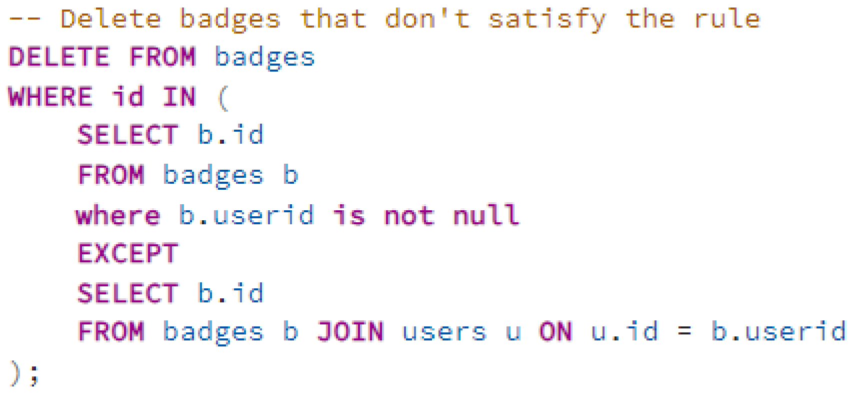
```
-- Delete postlinks that don't satisfy the rule
DELETE FROM postlinks
WHERE id IN (
    SELECT pl.id
    FROM postlinks pl
    where pl.relatedpostid is not null
    EXCEPT
    SELECT pl.id
    FROM postlinks pl JOIN posts p ON p.id = pl.relatedpostid
);
```

Completed in 379 ms (deleted 13674)



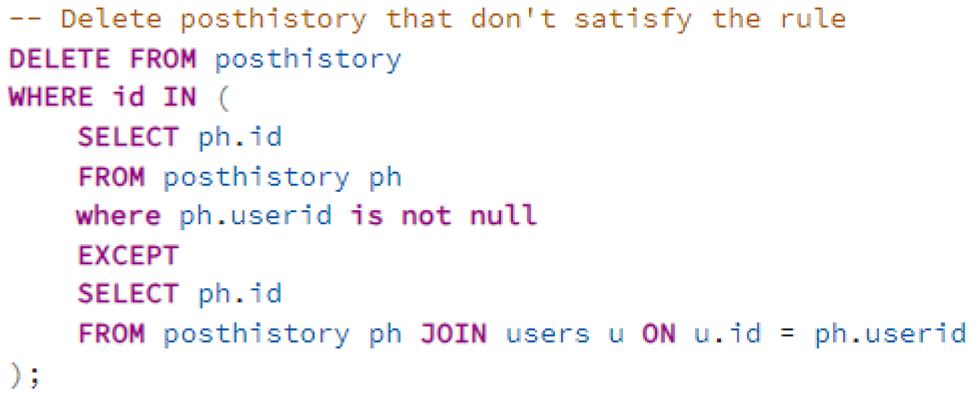
```
-- Delete votes that don't satisfy the rule
DELETE FROM votes
WHERE id IN (
    SELECT v.id
    FROM votes v
    where v.postid is not null
    EXCEPT
    SELECT v.id
    FROM votes v JOIN posts p ON p.id = v.postid
);
```

Completed in 11.025 seconds (deleted 254235)



```
-- Delete badges that don't satisfy the rule
DELETE FROM badges
WHERE id IN (
     SELECT b.id
     FROM badges b
     where b.userid is not null
     EXCEPT
     SELECT b.id
     FROM badges b JOIN users u ON u.id = b.userid
);
```

Completed in 4.530 seconds (deleted 0)



```
-- Delete posthistory that don't satisfy the rule
DELETE FROM posthistory
WHERE id IN (
     SELECT ph.id
     FROM posthistory ph
     where ph.userid is not null
     EXCEPT
     SELECT ph.id
     FROM posthistory ph JOIN users u ON u.id = ph.userid
);
```

Completed in 17.664 seconds (deleted 0)

Before proceeding to delete form the posts table, it is necessary to add the foreign keys first, otherwise other dependencies will break causing more loss of entire rows, rather than just a field (postid)

## Creation of the Foreign Keys (only for SQL)

The creation of the foreign keys had to be done in stages, in the first stages the following queries were executed:

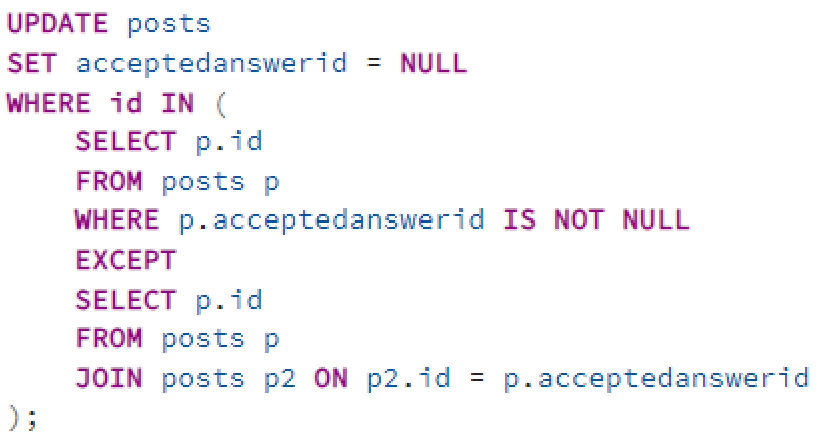
```
ALTER TABLE Comments
ADD CONSTRAINT FK_Comments_PostId FOREIGN KEY (PostId)
REFERENCES Posts(Id);

ALTER TABLE PostHistory
ADD CONSTRAINT FK_PostHistory_PostId FOREIGN KEY (PostId)
REFERENCES Posts(Id),
ADD CONSTRAINT FK_PostHistory_UserId FOREIGN KEY (userid)
REFERENCES Users(Id)
ON DELETE SET NULL;

ALTER TABLE PostLinks
ADD CONSTRAINT FK_PostLinks_PostId FOREIGN KEY (PostId)
REFERENCES Posts(Id),
ADD CONSTRAINT FK_PostLinks_RelatedPostId FOREIGN KEY (RelatedPostId)
REFERENCES Posts(Id);

ALTER TABLE Votes
ADD CONSTRAINT FK_Votes_PostId FOREIGN KEY (PostId)
REFERENCES Posts(Id),
ADD CONSTRAINT FK_Votes_UserId FOREIGN KEY (UserId)
REFERENCES Users(Id)
ON DELETE SET NULL;

ALTER TABLE badges
ADD CONSTRAINT FK_Badges_UserId FOREIGN KEY (userid)
REFERENCES Users(Id);
```

In 13.055 seconds

Finishing deletion from the posts table

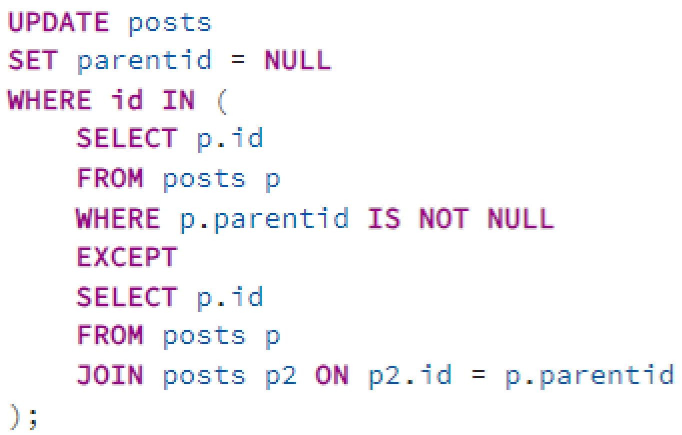
```
-- Delete posts that don't satisfy the rule
DELETE FROM posts
WHERE id IN (
    SELECT p.id
    FROM posts p
    where p.owneruserid is not null
    EXCEPT
    SELECT p.id
    FROM posts p JOIN users u ON u.id = p.owneruserid
);
```

Completed in 3.880 seconds (deleted 0)



```
UPDATE posts
SET acceptedanswerid = NULL
WHERE id IN (
    SELECT p.id
    FROM posts p
    WHERE p.acceptedanswerid IS NOT NULL
    EXCEPT
    SELECT p.id
    FROM posts p
    JOIN posts p2 ON p2.id = p.acceptedanswerid
);
```

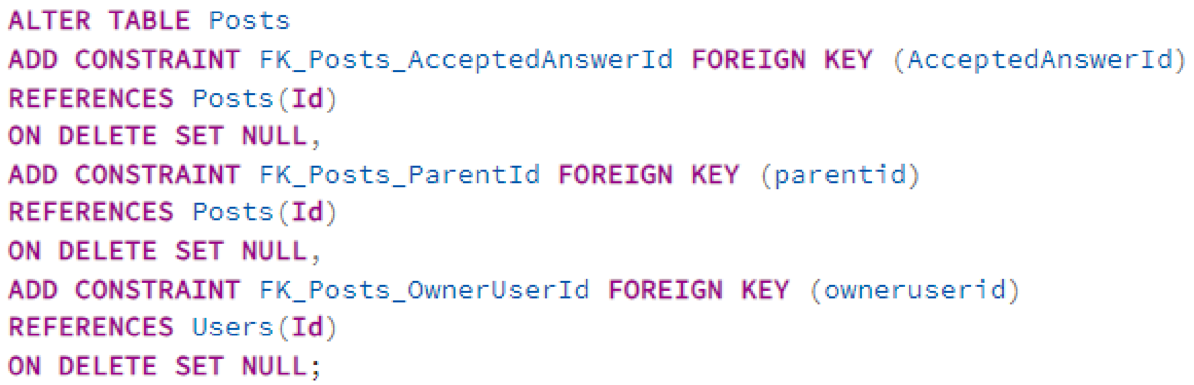
Completed in 2.726 seconds (updated 13)



```
UPDATE posts
SET parentid = NULL
WHERE id IN (
    SELECT p.id
    FROM posts p
    WHERE p.parentid IS NOT NULL
    EXCEPT
    SELECT p.id
    FROM posts p
    JOIN posts p2 ON p2.id = p.parentid
);
```

Completed in 12.574 seconds (updated 89487)

### Finishing addition of foreign keys



```
ALTER TABLE Posts
ADD CONSTRAINT FK_Posts_AcceptedAnswerId FOREIGN KEY (AcceptedAnswerId)
REFERENCES Posts(Id)
ON DELETE SET NULL,
ADD CONSTRAINT FK_Posts_ParentId FOREIGN KEY (parentid)
REFERENCES Posts(Id)
ON DELETE SET NULL,
ADD CONSTRAINT FK_Posts_OwnerUserId FOREIGN KEY (owneruserid)
REFERENCES Users(Id)
ON DELETE SET NULL;
```

in 7.191 seconds

## Cleaning the data in MongoDB

To keep the following comparison fair, we clean the same data from mongodb.
MongoDB compass doesn't seem to have a function to delete, and the query language doesn't really allow us to do it in one command.
We need to create python scripts in order to delete stuff or use the UI.

In order to time the operation (retrieval + deletion) we chose to use a python script (clean_data.py).
Result:

```
No invalid comments to delete.
Execution time for delete_invalid_comments: 28.57 seconds
No invalid posthistory records to delete.
Execution time for delete_invalid_posthistory: 47.57 seconds
Deleted 36953 invalid postlinks.
Execution time for delete_invalid_postlinks: 2.03 seconds
Deleted 2733 invalid postlinks based on relatedpostid.
Execution time for delete_invalid_postlinks_related: 1.46 seconds
Deleted 1110785 invalid votes.
Execution time for delete_invalid_votes: 87.51 seconds
Deleted 1 invalid badges.
Execution time for delete_invalid_badges: 32.03 seconds
Deleted 230871 invalid posthistory records based on userid.
Execution time for delete_invalid_posthistory_user: 52.29 seconds
Deleted 33962 invalid posts.
Execution time for delete_invalid_posts: 17.87 seconds
Updated 1012740 posts to nullify invalid accepted answers.
Execution time for update_accepted_answer: 40.33 seconds
Updated 523904 posts to nullify invalid parent IDs.
Execution time for update_parent_id: 31.42 seconds
Database connection closed.
```

# Different Types of Queries

| seconds | Reads | Write | Joins | Aggr | Text S | Nested Upd | Complx Joins | Pagination | Deletion | Count |
|---|---|---|---|---|---|---|---|---|---|---|
| SQL | 0.779 | 0.453 | 14.673 | 2.236 | 1.286 | 3.259 | 8.998 | 0.583 | >8h | 0.884 |
| MongoDB | 6.867 | 0.003 | 88.112 | 2.518 | 2.3296 | 1.7278 | >6h | 2.470 | 10.869 | 1.144 |

# Flexibility & Scalability: SQL vs. MongoDB

**Slide 1: Flexibility – SQL vs. MongoDB**

📌 **SQL (Relational Databases)**

- **Strict Schema**: Tables require predefined schemas, enforcing data consistency.

- **Normalization**: Data is structured across multiple tables, reducing redundancy but requiring complex joins.

- **ACID Compliance**: Ensures transactional integrity, making it ideal for banking and enterprise systems.

- **Changes Require Migrations**: Altering schemas (e.g., adding/removing columns) often requires downtime or migrations.

#### 📌 MongoDB (NoSQL)

- **Schema-less**: Uses flexible JSON-like documents (BSON), allowing dynamic fields.

- **Denormalized Structure**: Stores related data in embedded documents, reducing the need for joins.

- **Eventual Consistency**: Optimized for speed and availability rather than strict consistency.

- **Easier Schema Evolution**: Can handle unstructured and semi-structured data without requiring migrations.

🔍 **Key Takeaway**:

SQL is **rigid but reliable**, making it great for structured, well-defined data.

MongoDB is **flexible**, making it ideal for evolving applications, real-time analytics, and unstructured data.

### Slide 2: Scalability – SQL vs. MongoDB

#### 📌 SQL (Vertical Scaling)

- **Scales up** by increasing server resources (CPU, RAM, SSD).

- **Limited horizontal scaling**: Complex sharding and partitioning strategies are needed to distribute data across multiple nodes.

- **Joins & Transactions Make Scaling Harder**: Expensive queries can slow performance on large datasets.

- **Replication Support**: Provides **read scalability** via replication but **write scalability** is limited.

#### 📌 MongoDB (Horizontal Scaling)

- **Designed for Sharding**: Data is **automatically distributed** across multiple servers (shards).

- **Scales out** by adding more nodes, making it highly suitable for large-scale applications.

• **Eventual Consistency Model**: Allows high availability and fault tolerance at the cost of strict ACID guarantees.

• **Easier Global Distribution**: Supports geo-partitioning, where data is stored close to users for better performance.

🔍 **Key Takeaway**:

SQL databases scale **vertically**, which can be costly.

MongoDB scales **horizontally**, making it a better choice for **big data, high-traffic applications, and real-time analytics**.

Would you like a visual comparison table for your slides? 📊🚀