

# Tag Database Performance

David Glasser

Chris Lesniewski-Laas

Richard Tibbetts

December 15, 2006

## Abstract

“Tagging”, associating a set of words with an object, has become a ubiquitous feature in social Web applications. Most such applications use RDBMS backends, which struggle to provide the necessary performance for interactive tag browsing and search. As a result, we have observed that some developers have resorted to denormalized schemas or convoluted SQL queries for tag operations, instead of normalized relational operations that avoid anomalies and provide flexibility.

We set out to evaluate whether hacks were really necessary to optimize RDBMS performance on tagging workloads. We developed a test harness for various tagging implementations and found that, despite the fact that tagging seems to be a prototypical example of a relational many-to-many relation, it is in practice faster to use non-relational full-text queries on a denormalized table. We also attempted to diagnose the source of this performance anomaly, meeting with mixed success. Based on our results, we offer our best recommendations for application developers today.

## 1 Introduction

Organizing large quantities of information is a common problem. Historically, librarians and other like-minded people solved it with carefully constructed taxonomies and categorizations. This was sufficient when creating new data, such as books, was expensive, and the rate at which they were added to libraries was slow. Various techniques were developed to optimize increasingly complex categorization and discovery of items, such as topic maps.

The increasing popularity of digital cooperative publishing means that there are now collections of data growing too fast to be organized into top-down taxonomies by professionals. Additionally, there are many

smaller data sets which are produced and maintained by informal communities, where a professional librarian is not available. There is a need for a community-maintained organization structure that lends itself to non-professional contributions. Complex approaches like topic maps are unsuitable. Strict hierarchy is insufficiently flexible, and also not ideal for many contributors.

A popular solution to this problem is “tagging”. In a tag system, each item is associated with a set of tags. The tags are generally single words or short phrases. Tags have no inherent meaning: the reader must infer their meaning from context and experience. A tag with an ambiguous meaning might be disambiguated by additional tags on the item. For example, a picture tagged “windows” could be of a stained glass window in a church or a Microsoft screenshot; the former might also be tagged “church” and the latter “screenshot”. Users in a system like this assign tags to items, and search for other items by tag. The set of tags associated with an item should serve as a good description of that item.

Tagging is being used by a large number of web services to organize their data. Tagging is used by Flickr to manage photographs, del.icio.us to manage links, and YouTube to manage videos. Sites like Slashdot use tags to categorize user-submitted stories. More established services like Amazon and Yahoo have also begun to use tagging. In some of these systems, tags are global; in others tags are per-user. But all share a common core of functionality.

The basic operations in a tagging system are adding and removing tags from items and searching a set of items by their tags. There are a few common searches:

- Look up item by a given tag (“lookup”)
- Look up items with all tags in a given set (“intersection”)
- Look up items with all tags in a given set, without tags in another given set (“subtraction”)
- Find items which share many tags with a given item, and are thus presumed to be related (“related items”)
- Find tags which share many items with a given tag, and are thus presumed to be related (“related tags”)
- Enumerate popular tags (“tag cloud”)

These web applications are generally built on top of relational database management systems (RDBMS), using SQL as the interface between the programming language and the data store. Often an Object Relational Mapper (ORM) is used to abstract away the SQL and provide an application-level object model of the

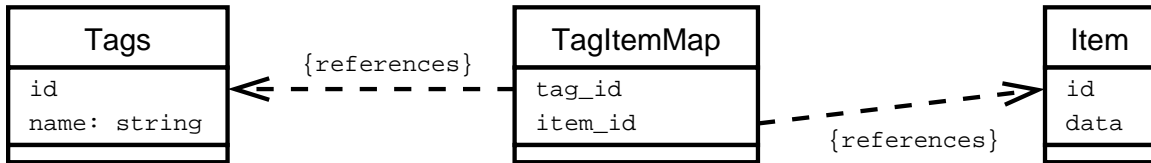


Figure 1: Normalized Tags Schema

data. Most web services use a free, open-source database such as MySQL, PostgreSQL, or SQLite as their underlying data store.

A data structure designed specially for tagging could generally have higher performance. RDBMSs are generally preferred because of features other than raw performance. First, they provide atomic actions (insert, delete, update) which simplify development of applications with many concurrent users. Second, they provide more physical and logical independence in the data model as it evolves over time than a customized data structure would. Third, ad hoc database queries improve debuggability and administration. Finally, the databases perform well enough that the cost of building a custom data store often outweighs the potential benefits over using a standard open-source RDBMS.

Because RDBMSs are the preferred development platform for these applications, the question arises of how to make them perform as best as possible. In particular, the tagging operations discussed above are often fundamental to the application, and because these applications need to cope with heavy interactive workloads, the performance of many small queries is very important.

The classic RDBMS solution for tagging is a normalized schema with three tables. This schema can be seen in Figure 1. The Tags table holds the textual representation of the tags, and a unique identifier. The Items table has a unique identifier for each taggable item, and some data such as a photograph, news story, or URL. The TagItemMap join table associates zero or more tags with each item and zero or more items with each tag. This normalized schema lends itself to straightforward SQL queries to lookup items by tag or set of tags. An example query can be seen in Figure 3.

Unfortunately, there is a perception in the web application development community that this system for storing and querying tags is too slow. [1, 2] Because of this, developers have devised a variety of hacks in order to improve performance, using techniques like denormalization and database-specific extensions to SQL.

Traditional RDBMSs should be able to provide the performance required by applications. In this paper we will explore why they do not, and what can be done about it. Section 2 discusses our approach to the

problem. Section 3 describes our methodology for measuring performance and the results. Section 4 presents recommendations for application and database developers. Finally, section 5 presents our conclusions about the causes of this problem and potential future work.

## 2 Goals

Our two main goals are to quantify the tagging problem and, if possible, to solve it. To achieve the first, we implemented a number of different tag database backends; we describe them and their advantages and disadvantages in Section 3. We measured and attempted to explain the differences between the performance of the different backends.

Our second goal is to give application developers the right tools to address their performance problems by showing a properly normalized RDBMS configuration with good performance, by modifying an RDBMS to perform well on the normalized schema, or by explaining how best to work around the problem if neither of the other options are practical.

## 3 Measurements

### 3.1 Methodology

To measure the relative performance of different tag database implementations, we implemented a test harness in Python which records the speed of tag lookup, tag intersection, and related items queries. As a baseline, we also measured the speed of row retrieval by primary key where applicable. For each tag database implementation, the harness loads a data set into the database and tests each query against the data set. The harness prepares each query and executes it many times (in the tests below, 100 repetitions) to eliminate query parse overhead and average out the cold start overhead and any random measurement noise.

The test harness can synthesize tag datasets of varying size, and can also import external datasets. We obtained an anonymized dump from the Hiveminder task-list service [3] to use in our tests. This dataset contains approximately 30,000 items (tasks), 7,500 tags, and 50,000 item-tag associations. Figure 2 shows that the distribution of tags per item in the Hiveminder data follows a power-law curve.

The tag database implementation can vary in a number of dimensions, including RDBMS back-end, schema, indices, and queries. These dimensions are not entirely independent: for example, full-text search is only built in to the MySQL back-end, and requires a denormalized schema with a full-text index.

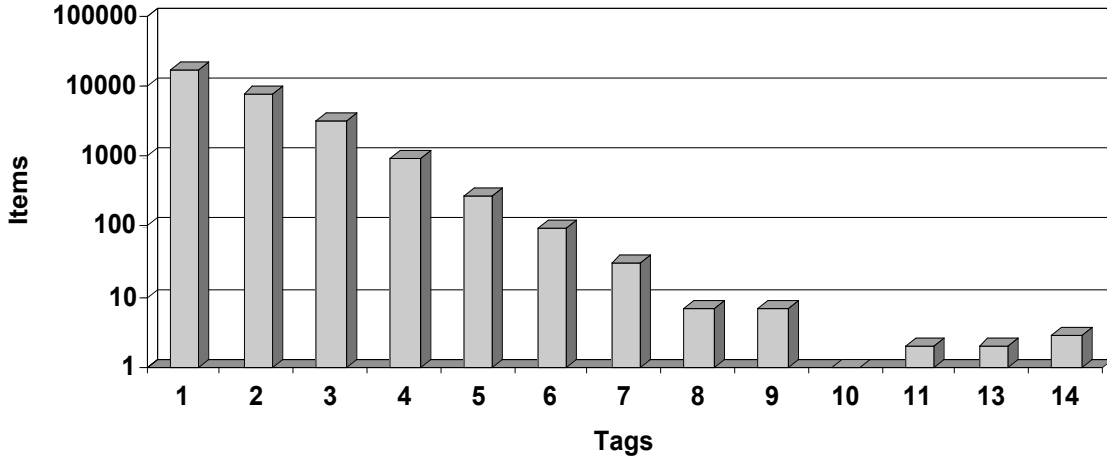


Figure 2: Distribution of tags in Hiveminder dataset.

Our test harness supports the MySQL, Postgres (7.4 and 8.1), and SQLite backends using the Python DB-API.<sup>1</sup> We also implemented support for a branch of Postgres called Bizgres which features bitmap indices. Finally, to get a sense of the overhead of using an RDBMS at all, we implemented a simple custom in-memory tag database using only Python’s native hash tables and set structures.

## 3.2 Schemas, indices, and queries

### 3.2.1 Normalized

Every RDBMS backend supports the standard normalized schema, with an items table, a tags table, and an item-tags map table. This schema has primary key indices on the ID columns of the items and tags tables, and a compound primary key index on both columns of the mapping table. To speed up tag-name-to-ID lookups, we construct an index on the name column of the tags table, and to speed up tag-to-item lookups and item-to-tag lookups, we constructed indices on both columns of the mapping table. We tested the performance of both B-tree and hash indices, although we don’t include the results below because we didn’t see any improvement over the default B-trees. We attempted to test bitmap indices in a branch of Postgres called Bizgres which supports them; however, we found that their use caused Bizgres to return incorrect answers.

The normalized schema supports several query styles which give equivalent result sets. The most straight-

<sup>1</sup>In theory, DB-API should make these interchangeable. In practice, the harness had to deal with many minor differences in both the API and the SQL layers.

```

SELECT Items.data
FROM Items
WHERE Items.id IN (SELECT TagItemMap.item_id
                   FROM TagItemMap, Tags
                   WHERE Tags.tag = 'sometag'
                   AND TagItemMap.tag_id = Tags.id);
AND Items.id IN (SELECT TagItemMap.item_id
                 FROM TagItemMap, Tags
                 WHERE Tags.tag = 'othertag'
                 AND TagItemMap.tag_id = Tags.id);

```

Figure 3: Intersect query using IN

```

SELECT Items.data
FROM Items i, ItemTagMap m1, ItemTagMap m2, Tags t1, Tags t2
WHERE i.id = m1.item_id AND i.id = m2.item_id
      AND m1.tag_id = t1.id AND m2.tag_id = t2.id
      AND t1.name = 'sometag' AND t2.name = 'othertag'

```

Figure 4: Intersect query using simple joins

```

SELECT Items.data
FROM Items, ItemTagMap, Tags
WHERE ItemTagMap.tag_id = Tags.id
      AND ItemTagMap.item_id = Items.id
      AND ( (Tags.name = 'sometag') OR (Tags.name = 'othertag') )
GROUP BY Items.id, Items.entry
HAVING count(Items.id) = 2

```

Figure 5: Intersect query using aggregation

```

CREATE TABLE fulltext_items (
  id int PRIMARY KEY,
  data text NOT NULL,
  tag_names text NOT NULL,
  FULLTEXT(tag_names)
)

```

Figure 6: MySQL denormalized items table

```

SELECT data
FROM fulltext_items
WHERE MATCH(tag_names)
      AGAINST('+sometag +othertag' IN BOOLEAN MODE)

```

Figure 7: MySQL denormalized intersection query

forward and readable uses the SQL IN operator (see Figure 3). Unrolling this query into an equivalent join-style query (Figure 4) improves performance on RDBMSs without robust query rewriting (such as MySQL). The tag-intersection query can also be implemented using aggregation instead of a second join (Figure 5), resulting in different query plans.

### 3.2.2 Denormalized MySQL with full-text search

MySQL’s built-in full-text search function, intended for natural language search of document corpora, can be repurposed for Boolean queries on tag databases such as lookup, intersection, and subtraction. Instead of (or in addition to) the normalized tables, the schema contains a denormalized items table, shown in Figure 6. Each row contains the data for a single item, plus an additional `tag_names` text column which contains all of the tags associated with the item, separated by spaces. The MySQL full-text index on this column enables searching for rows by tag. The SQL for an intersection query is shown in Figure 7: the `IN BOOLEAN MODE` modifier specifies a mode in which `+` and `-` prefixes indicate that the given word *must* or *must not* be present in the tags list.

This tag database implementation is problematic for multiple reasons. First, it is specific to the MySQL backend, and the matching algorithm is not fully specified by MySQL. Second, since the schema is not normalized, applications are likely to encounter data anomalies. For example, there is no way to have a record of a tag without any corresponding items; and removing a particular tag from every item would require an awkward combination of queries and application-side tag-list processing. Keeping track of the most popular tags would require keeping another side table, possibly leading to more anomalies.

In addition, the MySQL full-text index is not designed for this purpose. This shows up in various minor

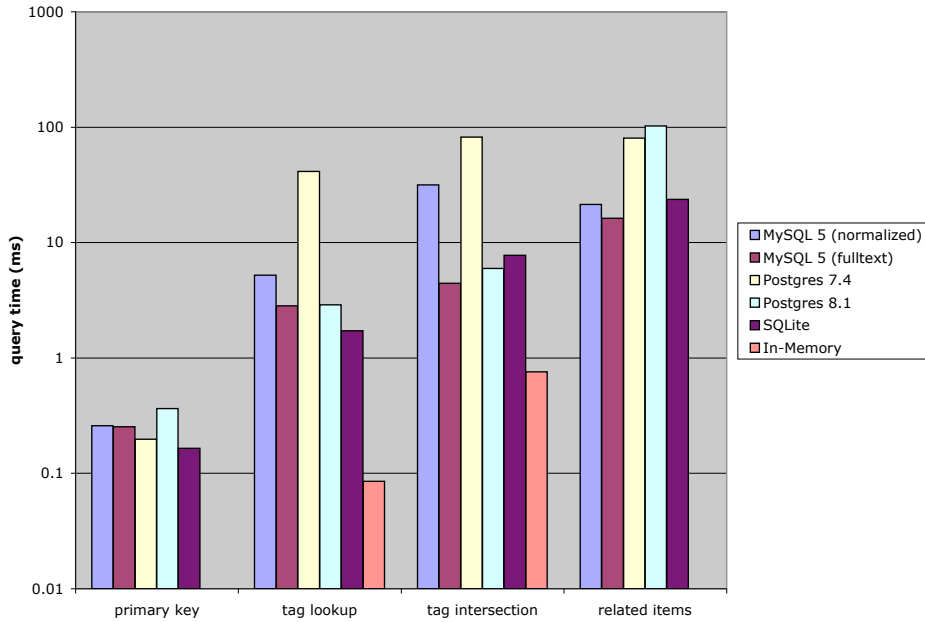


Figure 8: Time per query for several queries on several backends. (For normalized schemas, the best choice of query for each type was used.)

and annoying ways: for example, MySQL contains a hard-coded list of “stop words” which it ignores in queries. To have tags with these names, a developer either needs to recompile MySQL or to prefix every tag with some fixed string (to work around the stop-word list). Since the full-text search doesn’t support full relational queries, composing more advanced queries may become difficult. Popular-tags and related-items are both complicated for this reason.<sup>2</sup>

### 3.3 Results

Figures 8 and 9 show the time per query for various queries (row by primary key, items by tag, items by both of two tags, and related items) against various tag database implementations. Figure 8 shows the broad differences among backends and schemas. The typical RDBMS query took two orders of magnitude longer to execute than our custom in-memory implementation, so overhead was significant. Within the normalized schema, SQLite and Postgres 8.1 were perhaps an order of magnitude faster than MySQL and Postgres 7.4.<sup>3</sup> The MySQL full-text search had a significant edge, particularly on more complex queries. As mentioned above, the bitmap indices which inspired us to investigate Bizgres ended up being inaccurate.

<sup>2</sup>It’s possible to kludge related-items by using MySQL’s full-text ranking, which raises the rank of rows with more tag matches. However, this is made more frustrating by the lack of a specification for the ranking algorithm.

<sup>3</sup>MySQL was only able to achieve this performance with carefully optimized queries.

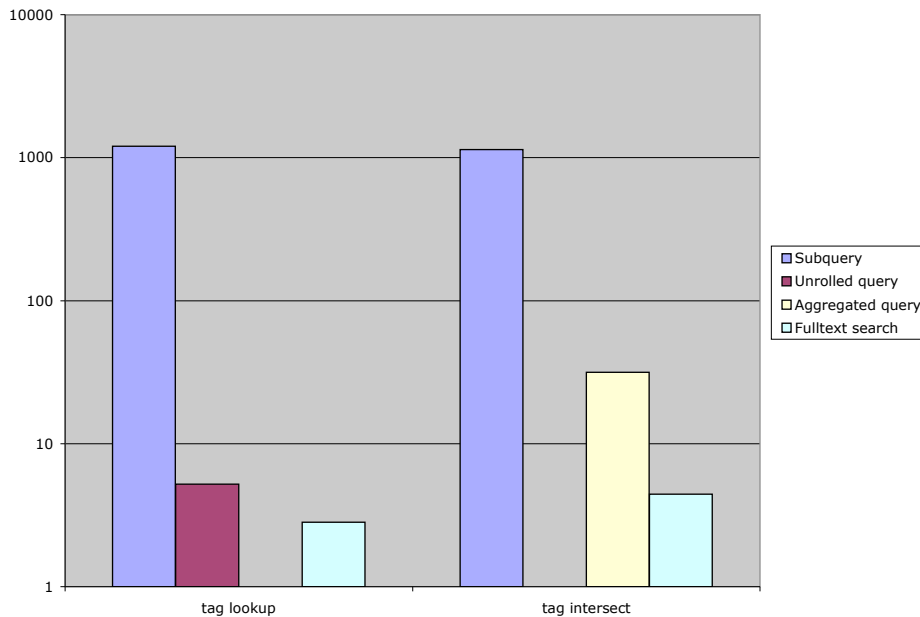


Figure 9: Time per query for several versions of two queries on MySQL. (Not all query types are applicable for both queries.)

Figure 9 shows MySQL’s performance with different query styles against the same normalized schema. MySQL executes the IN-style queries slowly because it does not rewrite the subquery into a join. However, if we manually unroll the join, MySQL’s performance is only a small factor worse than Postgres; and using the full-text index, MySQL consistently outperformed the alternatives.

## 4 Discussion

The high speed of the custom in-memory implementation, plus the small size of realistic tag data sets, suggests that the most practical solution for developers may be aggressive caching at the Object Relational Mapper interface. If most or all queries can be served out of this cache, treating the RDBMS as a slow backing store, the application will see orders of magnitude of performance improvement over direct SQL queries. However, this approach has the disadvantage of being tied to the specific ORM layer used by the application. If RDBMS performance could be improved instead, then all applications using the RDBMS could benefit, regardless of the language and ORM the application uses.

The improvement between Postgres 7.4 and Postgres 8.1 appears to be from the addition of auxiliary bitmaps to speed up equijoins. Unlike the (non-functional) bitmap indices in Bizgres, Postgres 8.1 creates

these join bitmaps on the fly and then releases them after the query is finished.

The denormalized schema using MySQL full-text queries was much faster than we initially expected. (In some tests on small data sets, we found full-text searches to be as fast or faster than lookup by primary key!) We pored over the source to MySQL’s full-text engine to try to figure out the cause. The engine contains many layers and code paths unrelated to our simple Boolean searches; after peeling all this away, we found that the underlying structure of the full-text search is not particularly high-tech. The full-text index is essentially an inverted B-tree index mapping words (tags) to rows (items), sorted by word first and row-ID second. When performing an intersection search, the access method looks up the set of rows corresponding to each word, and merge-joins the sorted row lists. To subtract, the access method simply filters out any rows matching the unwanted words.

Since this algorithm is essentially the same as the query plan produced by a join-style query on a normalized database, this doesn’t answer the question of why it is faster. The simple explanation is that most or all of the difference between the custom in-memory implementation (baseline) and the RDBMS queries is caused by various kinds of overhead, and that the full-text search avoids some of that overhead.

We tried to determine the sources of overhead in our join-style queries using oprofile and strace, but didn’t find a smoking gun. We ruled out disk accesses and syscall overhead using strace. Lock contention overhead seems unlikely, since only one RDBMS worker thread was running at a time; however, there might be some small cost to simply acquiring and releasing the uncontested locks. What remains is general CPU overhead: string comparisons and hashing (we verified that this took a non-negligible percentage of the time for Postgres), vtable overhead, CPU cache access patterns, and so on. The only broad conclusion we can draw is that the flexibility of general operators in query plans comes at a cost in query performance.

We also discovered that MySQL did not “unroll” subqueries such as the IN clause in Figure 3. This led to terrible performance on those queries: up to 200 times worse than the best MySQL time, with times of over a second just for single tag lookups.

## 5 Conclusions and looking forward

The goal of this project was to explore and solve the performance problems of Web application developers storing tag databases in RDBMS backends. We started with the expectation that a simple application of appropriate indexes would likely speed up the relevant queries without resorting to crude hacks like the MySQL full-text search over a denormalized table. To us, the most surprising result of our investigation

was that the Web developers weren't simply unsophisticated RDBMS users. Instead, we discovered that the denormalized MySQL schema is hard to beat.

The best advice we can currently offer depends on the developer's specific situation. The MySQL full-text hack may be appropriate if data anomalies aren't a big problem. Web applications are not like classical DB users: they typically are the only application with access to the database, and the code isn't distributed outside of the organization. If you control the database and its only client application, it's much easier to clean up the mess caused by data anomalies later, by decomposing the schema and modifying the queries appropriately.

If a normalized schema is desirable, you can do pretty well (only a few times slower) by using SQLite (faster) or Postgres 8.1 (more features). We would still like to figure out how to close the gap between our custom implementation and these RDBMS backends: there's no good excuse for the databases to be so slow, since this kind of join query is their main purpose. In theory, it ought to be possible to construct an analog of the MySQL full-text index for the normalized schema. However, in practice, this will require major hacking to construct and enable new indexes and access methods, so we may want to set our sights lower.

Finally, even if we construct a custom index which fits entirely in memory, we don't really expect to reach the performance of our custom implementation. Since developers are using Object-Relational Mappers anyway, it may be more practical today to improve the performance by caching aggressively at that layer, than to improve the performance of the RDBMS layer.<sup>4</sup>

## References

- [1] P. Keller. Tagsystems: performance tests. <http://www.pui.ch/phred/archives/2005/06/tagsystems-performance-tests.html>, June 2005.
- [2] T. O'Reilly. Database war stories #3: Flickr. [http://radar.oreilly.com/archives/2006/04/database\\_war\\_stories\\_3\\_flickr.html](http://radar.oreilly.com/archives/2006/04/database_war_stories_3_flickr.html), April 2006.
- [3] J. Vincent. Hiveminder (<http://hiveminder.com>) tag data. Personal communication, Nov 2006.

---

<sup>4</sup>We could develop a tool to do this, but that wouldn't really be a database project anymore.