Project Report - NullDB

Scenario

Marketer

1. Problem Specification

Users need a dashboard to interact with a database. Can we support a dashboard-- a set of queries presented in graphics/widgets-- efficiently? Design techniques to allow users to monitor data through a dashboard. In this topic, your focus is to make the recalculation/refreshing of the dashboard efficient. Our scenario is marketer.

2. Data

2.1 Data Source

We used Amazon product reviews, imported from Kaggle. The data is in a CSV format. Each row represents a product review, consisting of the following headers: id, dateAdded, dateUpdated, name, asins, brand, categories, primaryCategories, imageURLs, keys, manufacturer, manufacturerNumber, reviews.date, reviews.dateSeen, reviews.didPurchase, reviews.doRecommend, reviews.id, reviews.numHelpful, reviews.rating, reviews.sourceURLs, reviews.text, reviews.title, reviews.username, and sourceURLs.

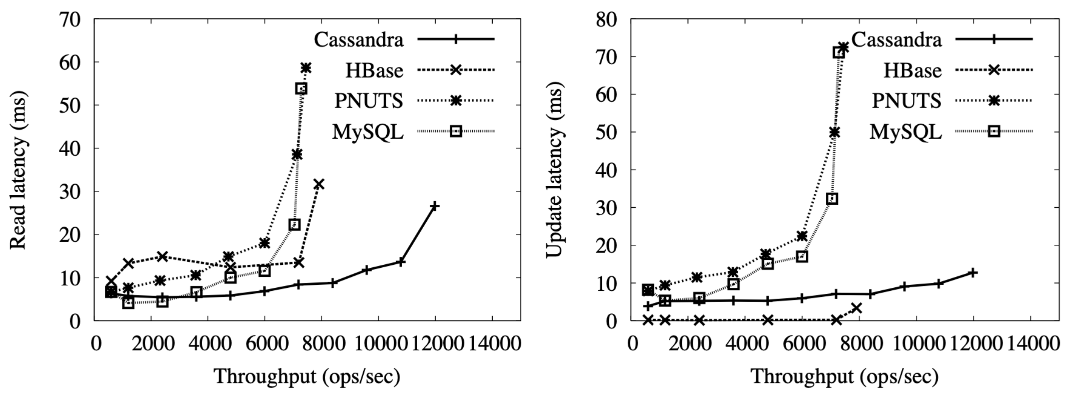
2.2 Database Options

The choice of database is dependent on many factors. We considered the following factors for an appropriate choice of the database. These are: read/write latency, transactions, concurrency, consistency, availability and partition tolerance.

The choice of the features are decided from our use-case. The read/write latency of a database is important since it directly impacts insertion/query time. Since we only have a single database instance, there isn't a need to support a high number of concurrent connections. Instead we can handle concurrent connections on the server, thereby using a single database connection to connect to the database.

The input dataset is not incredibly large (roughly 30,000 entries), but our design allows for scalability, which we stress as a critical component in our design. Due to the nature of the problem, we need strong consistency and availability guarantees. We aim to minimize the downtime, and thus sacrifice partition tolerance.

We use the base results presented by Brain Cooper et. al.[2], Rachit Pandey et. al.[3] and Yuqing Zhu et. al.[4] in addition to our benchmarks to ratify our decisions.



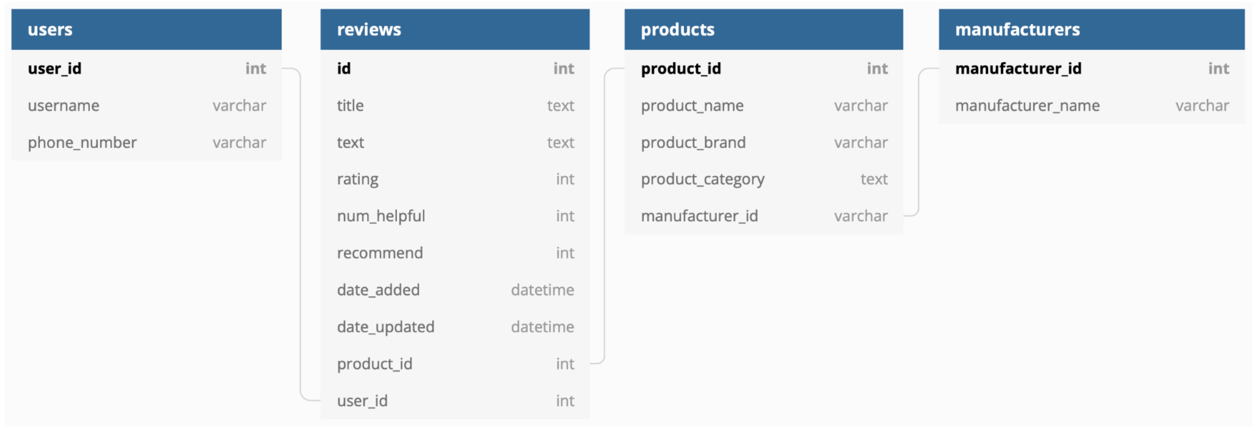
In low-throughput read-workload applications, MySQL shows the best performance compared to all the other databases. In low-throughput write-workload applications, MySQL has the second best performance, second only to HBase. Since our workload is exceptionally read-heavy, MySQL is a suitable choice

We also used local YCSB benchmarking data results to analyze latency. We came to the conclusion that our database would need to cater to consistency and high availability while compromising on partition tolerance. On similar lines, we also need support for transactions to ensure strong consistency guarantees. The alternative is to not have transactional guarantees and handle consistency on the user-end, which will likely result in unnecessary overhead given the size of the input dataset.

Relational databases also have a fixed schema as opposed to NoSQL databases where the schema is semi-structured or unstructured. Since our data is inherently well-structured, we opted for a relational database. Our choice is also driven by the hardware resources available.

2.3 Database Schema

The database schema is normalized to 3NF to minimize data redundancy and maintain data integrity. A significant majority of production-level databases are in 3NF; therefore the project replicates it to better simulate real-world environment/workload. Further normalization (BCNF, 4NF, 5NF) degrades system performance without yielding significant data model benefits.[1]



users	
user_id	int
username	varchar
phone_number	varchar

reviews	
id	int
title	text
text	text
rating	int
num_helpful	int
recommend	int
date_added	datetime
date_updated	datetime
product_id	int
user_id	int

products	
product_id	int
product_name	varchar
product_brand	varchar
product_category	text
manufacturer_id	varchar

manufacturers	
manufacturer_id	int
manufacturer_name	varchar

3. Work

We developed a full-stack web application using React.js, Python Flask, and MySQL as proof of concept.

3.1 Widgets

Business owners can view the top rated products and top recommended products to make better decisions regarding which items to display. Owners can also view the manufacturers with the highest average product rating to make better decisions regarding where to source new products. The most active users widget tracks user engagement, facilitating the development of rewards and loyalty programs.

Users can select the corresponding product to view the most helpful and most recent reviews for a given product. Users can interact with reviews in live time. By upvoting certain reviews, the change will reflect upon other user dashboards.

3.2 Improvements/Innovations

In the naive implementation, the query is executed on the base relations in each update. The reviews table is relatively large, and thus it is incredibly inefficient to constantly re-query the entire dataset with each update. We turn to the techniques introduced by Ahmad et al.[5] and Gupta and Mumick[6] to optimize query performance.

The algorithm we implement enables efficient dashboard updates. Prior to ingestion, we initialize a set of counters with default values, also known as a summary table. The counters are an aggregated map of the mappings: <product_id, product_name, rating> and <product_id, product_name, recommended>. We define a database trigger to update the counters accordingly. A database trigger is a special stored procedure which runs when specific actions occur in a database. In local testing, database triggers prove to be slightly faster than manually issuing a separate SQL instruction after INSERT. We assume there are significantly more reviews than products. With the technique above, queries need only access the summary table, which consists of COUNT(products) entries instead of COUNT(reviews) entries. The query executes against a limited subset of the original table orders of magnitude smaller than the base table. By storing the product name alongside the product_id, we can also avoid an expensive INNER JOIN operation.

With recent advancements in memory technology, it is possible to store the summary table in cache. We turn to the techniques introduced by Debnath et al.[7] and Fan et al.[8] to further optimize performance. We exploit the method of building an in-memory caching model through a custom hash table. The caching model is persistent and is built to handle high throughputs. In terms of the higher-level model, we employ a write-back cache with a custom eviction policy.

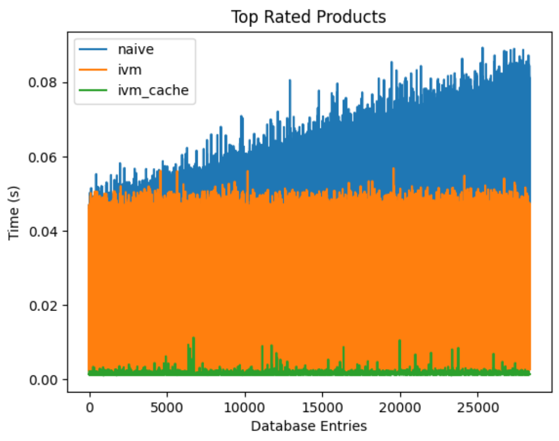
Products with the lowest aggregates are evicted when the cache fills up. The summary table is thus stored and modified in the server instead of the database, thereby increasing performance.

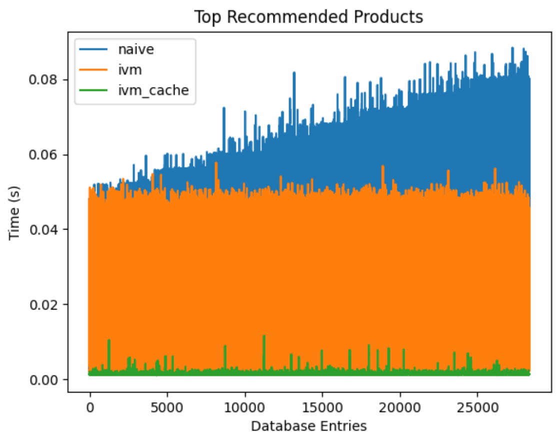
To further minimize the data-to-query lag, we optimize for insertions as well. It takes time to establish a connection to the database to insert a single record. If there is a sudden influx of records (importing a set of existing reviews) the server may be bottlenecked by insert times. Therefore, it is necessary to support some form of additional processing to minimize server overload and maintain realtime-ness. We turn to the techniques introduced by He et al.[9] to further optimize performance. By incorporating the concept of batch insertions with our current optimizations, we are able to support significantly faster data flows without issue. The database triggers and caching mechanism are updated accordingly to maintain support for existing features.

4. Results

4.1 Query Benchmarks

We benchmark the top rated products and top recommended products after each database insertion. The query response times are shown below. Each graph plots the respective query time against the number of elements in the database.



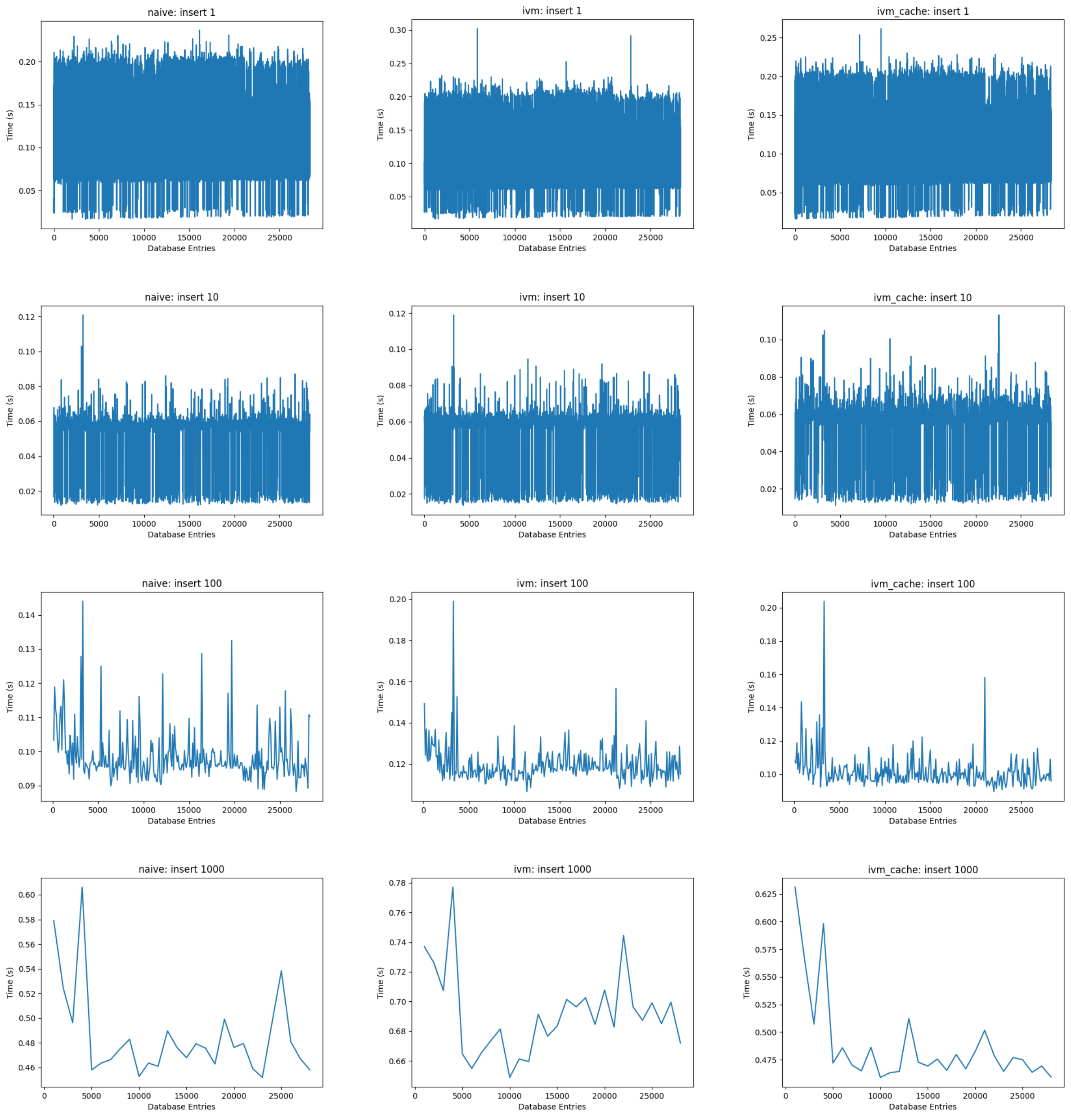


In the naive implementation, the query response time is directly proportional to the number of database entries. The query aggregates across the entire review dataset, which can be an extremely expensive operation in large-scale databases. As the number of reviews increase, the query time increases as well. The approach isn't well suited for real-world applications as it doesn't scale efficiently.

In the incremental view maintenance implementation, the query response time remains relatively constant regardless of the number of database entries. The number of elements in the summary table scales proportionally to the number of products instead of the number of reviews in the database. Query time is dominated by the propagation delay and transmit time of the network. The approach can be further optimized by caching the summary table in the server. With recent advancements in memory technology, it is feasible to store the summary table in main memory, thereby eliminating the database access completely. Network latency/bandwidth is rendered mute.

4.2 Insert Benchmarks

We benchmark single insertions and batch insertions of varying sizes (10, 100, 1000) across configurations. The insert times are shown below. Each graph plots the respective insert time against the number of elements in the database. The graphs are ordered such that we can compare different configurations and batch sizes across axises.



The insert time for the incremental view maintenance implementation is slightly higher than the insert time for the naive and incremental view maintenance + cache implementations. When a new review is inserted into the database, the database executes a stored procedure to update the summary tables. The difference in insert time can likely be attributed to the additional operations on disk.

In the incremental view maintenance + cache implementation, the summary tables are likewise updated on new reviews. However, as the information is stored in main memory, the operations execute significantly faster, having minimal effects on overall performance.

It is clear batch processing increases performance for significant amounts of data. As we increase batch size, the amount of time it takes to insert *n* elements is significantly less than the amount of time it takes to insert *n / k* elements *k* times. For example, in the naive case, it generally takes roughly 0.2 to 0.25 seconds to insert one element, whereas it takes roughly 0.45 to 0.6 seconds to insert 1000 elements. The trend holds true for all other configurations. Network propagation delay dominates transfer time, so it is beneficial to transfer as much data as possible over a single connection rather than opening a new connection for each piece of data. The performance benefits will likely taper off at some point, but we currently lack the data to scale the batch size further.

5. Conclusion

The incremental view maintenance + cache implementation provides noticeable improvements to query performance when compared to a naive implementation. Query times scale to COUNT(products) instead of COUNT(reviews), which is typically orders of magnitude smaller. The inclusion of batch processing enables it to support rapid data streams without sacrificing reliability or realtime-ness.

While our model performs quite well, it is open to further improvements. Firstly, we can improve on the current in-house indexing scheme and instead build a new indexing scheme by basing it on pre-existing hashing techniques. We can also try to scale out this model to handle concurrency on a larger scale. Furthermore, we can look at ways to solve this problem for nested aggregate queries, also called non-monotonic queries, to support a wider range of information.

6. References

1. Lee, H. (1995). Justifying database normalization: a cost/benefit model. Information Processing & Management, 31(1), 59–67. doi:10.1016/0306-4573(95)80006-F
2. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. *Benchmarking cloud serving systems with YCSB*. In Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10). Association for Computing Machinery, New York, NY, USA, 143–154. DOI:https://doi.org/10.1145/1807128.1807152
3. Rachit Pandey. *Performance Benchmarking and Comparison of Cloud-Based Databases MongoDB (NoSQL) Vs MySQL (Relational) using YCSB.*
4. Zhu Y. et al. (2014) *BigOP: Generating Comprehensive Big Data Workloads as a Benchmarking Framework.* In: Bhowmick S.S., Dyreson C.E., Jensen C.S., Lee M.L., Muliantara A., Thalheim B. (eds) Database Systems for Advanced Applications. DASFAA 2014. Lecture Notes in Computer Science, vol 8422. Springer, Cham. https://doi.org/10.1007/978-3-319-05813-9_32
5. Ahmad, Yanif & Kennedy, Oliver & Koch, Christoph & Nikolic, Milos. (2012). DBToaster: higher-order delta processing for dynamic, frequently fresh views. Proceedings of the VLDB Endowment. 5. 968-979. 10.14778/2336664.2336670.
6. Gupta, H., & Mumick, I. S. (2006). Incremental maintenance of aggregate and outerjoin expressions. Information Systems, 31(6), 435–464. doi:10.1016/j.is.2004.11.011
7. Debnath, B., Sengupta, S., & Li, J. (2010). FlashStore: High Throughput Persistent Key-Value Store. Proc. VLDB Endow., 3(1–2), 1414–1425. doi:10.14778/1920841.1921015
8. Fan, B., Andersen, D. G., & Kaminsky, M. (2013). MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, 371–384. Presented at the Lombard, IL. USA: USENIX Association.
9. He, H., Xie, J., Yang, J., & Yu, H. (2005). Asymmetric batch incremental view maintenance. 21st International Conference on Data Engineering (ICDE'05), 106-117.