DSC-204A: Scalable Data Systems, Winter 2024

15: Cloud Storage - 2

Lecturer: Hao Zhang *Scribe: Pragna Elavarthi, Pratishtha Gaur, Heejoo Shin*

1 Introduction - Previous Lecture Recap

The lecture started with a short recap of what was discussed in the previous lecture in which we discussed data indexes.

1.1 Straw-man Design

We tried to design a database from scratch, started with the Straw-man solution.
There are two functions: **DBSet** and **DBGet** which access each line as a key and a value and then we can continue to append to it.

Advantages and Disadvantages:
Writing operation in this particular design is pretty fast but the read operation is really slow because it is O(1).
As it is append-only it the storage space is very large.

1.2 Hash table

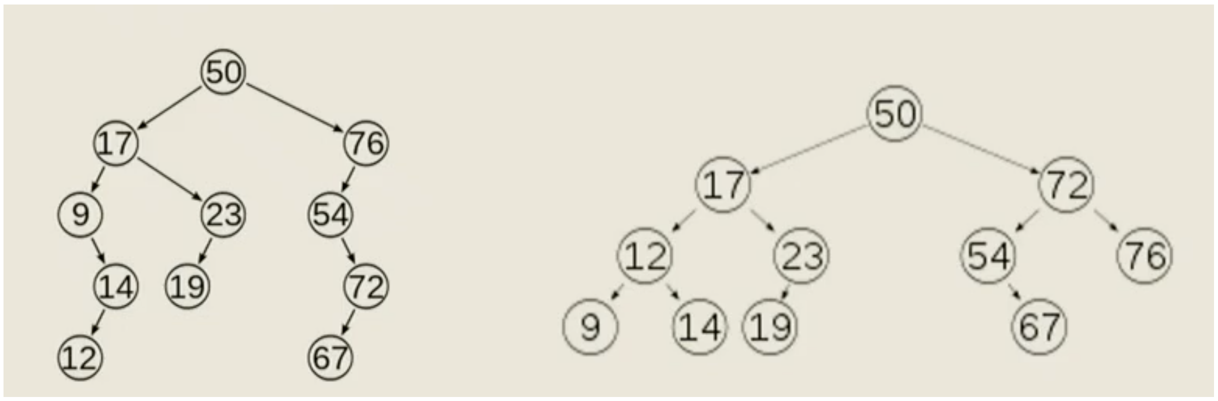
Improvisation to Straw-man design, stores all keys in memory and all values in the disk.

Advantages and Disadvantages:
Fast write and read operations, by querying the key, you can find the value so read operation also fast as compared to the previous one.
Needs less storage space.
Keys needs to fit in the memory.
Background compaction which uses merge sort can be done.

2 LSMTable : Augmented SSTable with MEMTable

Today's class discusses how to get these tables sorted. One approach is to add back-end processes to sort data on disk, but this is slow due to the lack of random access. As a result, sorting large databases becomes increasingly sluggish. The focus is on exploring strategies for efficiently sorting tables initially.
Next concept is integrating a Memtable into the LSM (Log-Structured Merge) table architecture, alongside the existing components, SSTable and LSM table. The Memtable, as an in-memory data structure, serves

as a temporary storage for unsorted data. Its presence capitalizes on the inherent advantages of memory for rapid random access, facilitating efficient sorting operations. By incorporating the Memtable, the system enhances its ability to manage data, ensuring that sorting operations remain optimized without the need for frequent recalculations. This augmentation streamlines the process of handling incoming records and managing historical data within the LSM table. Overall, the integration of the Memtable complements the existing SSTable and LSM table components, fortifying the system's performance and scalability in managing sorted data structures.
This uses self-balanced tree, which may be familiar to many as the binary search tree with a self-balancing mechanism. The difference lies in the self-balanced tree's ability to automatically maintain its height, ensuring it remains close to logarithmic. Essentially, this means that the tree strives to balance itself, minimizing height discrepancies. The self-balancing process involves occasional restructuring of the tree, known as tree rotations, to prevent the height from exceeding the logarithmic limit. It's a straightforward yet effective improvement over the basic binary search tree. However, why choose a self-balanced tree over a regular one? The answer lies in mitigating worst-case scenarios.



In traditional binary search trees, worst-case scenarios can occur due to imbalanced trees, resulting in operations taking linear time. By opting for a self-balancing tree, such as the Red-Black tree or AVL tree, we ensure that operations remain logarithmic in all cases, providing faster insertion and retrieval.

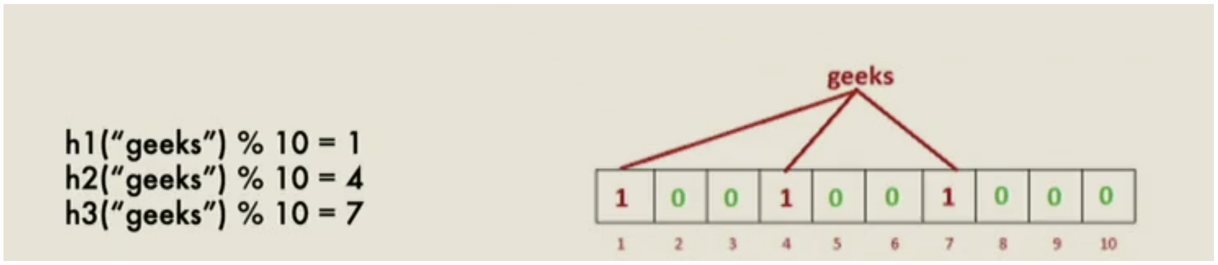
Now, how does this tie into our existing structure? Well, in our log-structured merge (LSM) table, incoming writes are directed to the Memtable, which maintains a self-balanced tree in memory. This tree continuously accommodates new entries while balancing itself to uphold efficient search operations. When the Memtable reaches a certain threshold size, known as the stretch threshold, it's flushed to disk, becoming the most recent segment. For reads, the system checks the Memtable first. If the desired key isn't found, it scans through the segments on disk, benefiting from their pre-sorted nature.

However, there's a potential issue with sparse indexing in memory. When querying for keys that may not exist in the database, the system first checks the Memtable, then scans through disk segments sequentially. This process can become slow if the key isn't found, resulting in scanning all segments unnecessarily. To address this inefficiency, we need to optimize our search strategy. Instead of blindly scanning through all segments on disk after checking the Memtable, we could implement a more intelligent lookup approach. This would involve refining our search algorithm to minimize unnecessary disk scans, thereby enhancing overall query performance.

3 Bloom filter

Let's delve into a powerful addition to our arsenal: the Bloom filter. Familiar with it? It's a nifty probabilistic data structure, ideal for quickly checking membership in a given set, such as a database. Here's the gist:

The Bloom filter offers space-efficient testing for set membership, where a simple query determines if an element belongs to the set. In our case, the set represents our database, and the query is akin to a database lookup. The computation involved is minimal, and space complexity is O(M). What's M? In a Bloom filter, M refers to the number of bits in the filter, which we've set to 10 in this instance. Additionally, we use three hash functions: H1, H2, and H3.
To check if an element, say "geeks" is in our database, we hash it using these three functions, yielding three



values: 1, 4, and 7. Then, we set the corresponding entries in our Bloom filter to 1.

This process is lightning-fast since it only involves three hash functions and simple memory operations. Now, to verify if an element exists in the database, we repeat this procedure. If the Bloom filter indicates that all corresponding positions are 1, it suggests the element might be in the database. However, there's a catch.

Bloom filters come with a trade-off: they might produce false positives. In other words, if the filter indicates an element is in the database, there's a probability it isn't. However, if it says the element isn't in the database, you can trust it. This reliability makes Bloom filters efficient for avoiding unnecessary queries to the entire database.

To improve accuracy, you can increase the number of hash functions and expand the number of entries. Although this adds computational overhead, it reduces false positives, enhancing the filter's effectiveness.

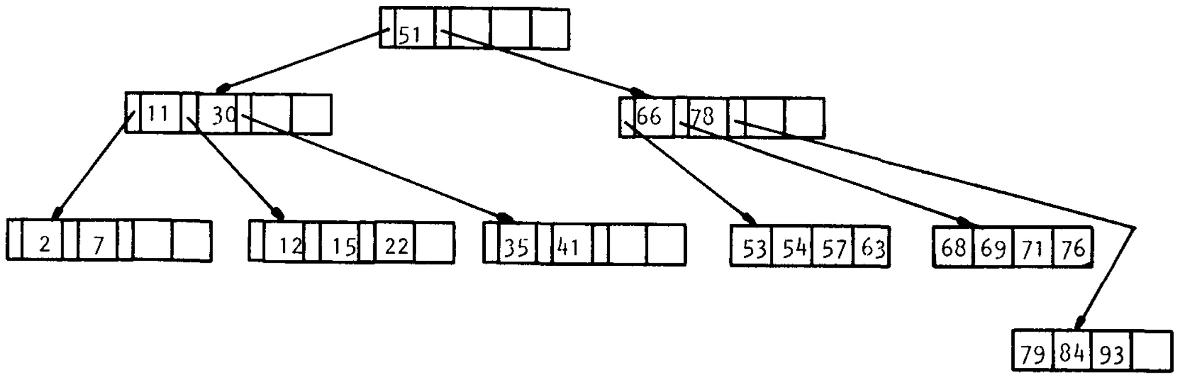
In summary, while Bloom filters offer speedy membership checks with minimal space requirements, they come with the caveat of potential false positives. Yet, their ability to confidently rule out non-membership makes them invaluable for optimizing database queries.

This addition enhances our log-structured merge table, providing an additional mechanism for efficient existence checks.

4 BTree

BTree is another prominent data storage method. It is a self balanced tree that means the height of subtrees of any node differ by at most one. This ensures that the tree remains relatively balanced, preventing one branch of the tree from becoming significantly longer than others. The only difference between BTree and a standard self balanced tree is that each node in a B-tree contains an array of keys and corresponding pointers to child nodes.

The number of elements in the array is called branching factor. This factor determines the maximum number of keys a node can hold and thus impacts the structure and performance of the tree. If more nodes then we can control the height since breadth will be higher so it will have more branching effect. We can reduce the height of the tree by increasing the branching factor.



The root node is stored in the memory and its children on the disk. We first search for the element in the root array if not found we can a segment of data, the page from disc and return the value. Similarly to append a new element the correct position is searched and replaced. The value is directly changed in the page.

4.1 Issues with BTree

When utilizing a B-tree with a fixed branching factor, such as 100, efficient insertion and maintenance of balance are crucial for optimal performance. However, inserting elements in ascending order can lead to significant challenges, resulting in **suboptimal space utilization** and reduced efficiency over time.

When elements are inserted in ascending order, new elements tend to be added to the rightmost side of the tree, causing certain nodes to become denser while others remain relatively sparse. The B-tree nodes may become unevenly distributed, causing certain nodes to fill up faster than others. This uneven distribution can lead to wasted space and **fragmentation** within the tree structure. Eventually, the tree may require page splitting to maintain its self-balancing property, but if the insertion pattern continues to be ascending, the splitting process may exacerbate the uneven distribution issue.

5 Trends: In-Memory Database

BTrees are are very old and were more useful in relational databases. But now we are trying to move to in-memory database. we are trying to avoid using disk and store database in memory.

One of the primary motivations behind the move to in-memory databases is the inherent speed advantage of RAM over disk storage. While disks are durable and cost-effective, their read/write speeds are relatively slow compared to RAM, and they do not support efficient random access. As RAM has become cheaper and more abundant, especially with the proliferation of deep learning instances offering vast amounts of memory, leveraging this resource for database operations has become increasingly feasible.

Furthermore, emerging hardware innovations, such as battery-powered RAM, promise to make RAM persistent, ensuring that data stored in memory is retained even during power outages or system reboots. This addresses one of the key concerns regarding the volatility of memory-based storage.

However, the transition to in-memory databases is not solely about speed. While RAM inherently offers faster access times, the way data is stored and processed in memory also plays a crucial role. Unlike disk storage, which often involves data compression to conserve space, data stored in memory remains in its native, usable formats, such as floats or other data types. This eliminates the need for cumbersome serialization and deserialization processes, reducing overhead and enhancing overall efficiency.

Despite the slightly higher cost associated with in-memory storage compared to disk-based storage, the benefits in terms of performance, convenience, and reduced overhead make it a compelling choice for many applications. There are many in-memory database implementations today like, Memcached, MemSQL, Oracle TimesTen, Redis.

6 Data Warehouse and Column Storage

The final section of the lecture discusses recent trends in databases: data warehouses and column storage. These technologies represent a huge leap in database technologies from traditional relational databases to cloud databases.

6.1 OLTP vs OLAP

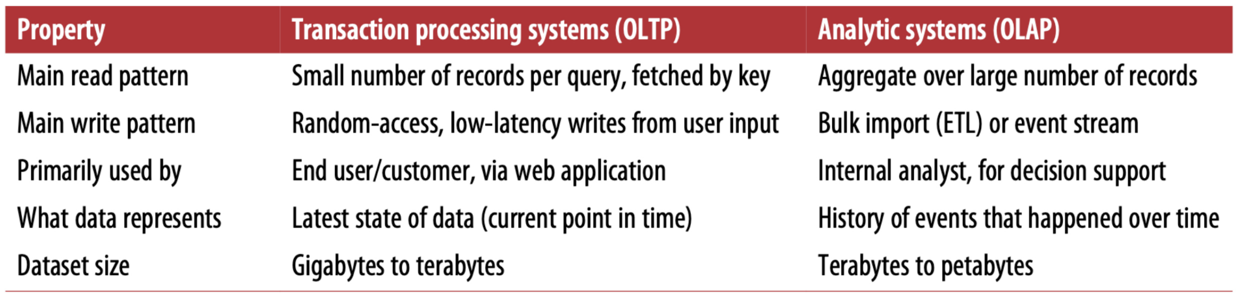
The majority of modern internet and mobile applications can basically be summarized to be doing CRUD: Create, Read, Update, and Delete. For example leaving a like or comment on a Youtube video translates into a CRUD operation inside the application. This leads us to a concept that has been critical in relational databases: **Online Transaction Processing (OLTP)**.

Many actions boil down to database transactions. To name a few, making sales, placing an order, employee payroll, blog post comments, user actions in games, adding/removing contacts to address books are all dispatched as database transactions. Processing these transactions are the field of OLTP.

However more recently people have become interested in data analytics, where we gain insights and values from data. A perhaps overused example is the Walmart Beer and Diaper, where Walmart discovered that placing diapers next to beer resulted into boosted sales of both items. In data analytics we are interested in finding such unexpected correlations. We ask questions such as *"What was the total revenue of our stores in January?"*, in which we are querying a range of historical data. Other possible questions are *"How many more bananas did we sell than usual for recent data?"* and *"Which brand of baby food is most often purchased together with brand X diapers?"*. Answering data analytics questions is basically the field of **Online Analytic Processing (OLAP)**.

There has been a debate between experts on whether databases should be designed for OLTP or OLAP–and the conclusion for now is to design separate databases for each of these purposes.

Figure 1 summarizes the differences between OLTP and OLAP. Additional points to note are that latency is not so important for OLAP writes, and that OLTP is often done for B2C applications while OLAP is often related to B2B.



Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

Figure 1: Comparison of OLTP and OLAP

6.2 Data Warehousing

Data warehousing refers to a new way of storing data for OLAP. Unlike OLTP, OLAP is primarily concerned with reading data for analytic purposes, and often involves copying a relational DB for read-only purposes. The motivation for data warehouses come from the complexity of transaction systems. Transaction systems require low latency and high availability. It is difficult to query ranges or perform analysis on these systems because it risks interrupting other transactions being performed in the system.

That is why we need a separate database, a data warehouse which analysts can query without worrying about possible interference with OLTP operations. This is a read-only copy of the database for analytic purposes. Data warehouses usually only exist for large companies, as small companies often lack big data from which to perform analysis. For example, small companies such as Levels.fyi have been known to have scaled to millions of users using Google Sheets as a database.

An important concept related to data warehouses is **Extract-Transform-Load (ETL)**. This refers to the process of transforming a relational database to a data warehouse. First we have *Extract*, where data is read periodically from the relational database and dumped into the OLAP database. Next *Transform* refers to transforming the relational DB's schema into one more suitable for OLAP and applying necessary data cleaning. Finally in the *Load* phase the data is loaded into a data warehouse.

Here we recap the motivations for data warehouses. Implementing data warehouses helps separate the different concerns of OLTP and OLAP databases. OLTP databases are more focused on performance in terms of reliability and latency, and we do not want data analysis to interrupt OLTP operations. Also separating the two results in better management of expertise, as data scientists have more expertise in dealing with data warehouses, while relational databases are better managed by system engineers. Furthermore the database indices discussed previously in the lecture (SSTable, B-tree) are good for reading and writing single records, but are not good for answering analytic queries that span over a large range of records.

There is one way to interact with both OLAP and OLTP: SQL queries. However OLAP often requires additional mathematical operations to be performed on top of the results of SQL queries, sometimes for distributed settings. This often cannot be done in pure SQL, and requires extra programming effort. This leads to a huge prospective market in the field of OLAP: how to provide better functionality for these type of tasks. Examples of OLAP software are Spark and Ray. In Ray, data is stored in-memory and can be fully coded in Python; unlike how previously end users needed to access Redis nodes.

OLAP is a multi-million dollar industry, and there have been many attempts in industry to provide a GUI-based, or otherwise user-friendly interface for OLAP. A few examples are Trifacta (sold for $400M), a company that designed a GUI analytics tool for CSV files, and Streamit (sold for $800M), which provided a high-level visualization interface. Databricks (valuated at $43B) provides data warehousing along with a Python notebook which provides functionality to process petabytes of data. Finally there is Snowflake (valuated at \$73B, and one of the largest public companies up to date) which similarly provides data warehousing along with a nice GUI to perform SQL queries.