NTNU

Kunnskap for en bedre verden

TDT4225 - Very Large, Distributed Data Volumes

Exercise 1

Author:
Martin Skatvedt

Date 19.09.2023

1. SSD

The Flash Translation Level is a firmware layer to prevent wear on SSD blocks, by implementing wear leveling. We want to distribute write tasks to all blocks uniformly, so that no blocks fail before others. The FTL is also responsible for garbage collecting. It goes through old blocks which are considered eligible and erases them. Deciding which blocks are invalid is defined by the garbage collector algorithm but can be something as x% of the pages in the block are invalid. Lastly the FTL keeps a mapping between logical block adresses (user/host adress space) and the physical block addresses.

2. SSD

Sequential writes generally provider better perfomance on all drives but SSD specifically. Random writes on SSDs creates internal fragmentation which reduces performance. Sequential writes also makes sure that the block distribution is even and prolongs the lifetime of an SSD. One important factor is that if we only use sequential writes, there is no need for complex garbage collection, since all blocks get filled up sequentially, the garbage collector only has to invalidate block by block as the writes proceed.

3. SSD

Writes which ware the same size as a multiple of a clustered page creates no more write overhead, however if its not the SSD controller needs to look up the content in the last clustered page and combine it with the new data before writing. If all write requests are equal to or a multiple of a clustered page, both sequential and random writes can perform at the same level. However if the write is bigger or smaller than a multiple of a clustered page, random writes will create external fragmentation. THis is opposed to sequential writes which will still fill up blocks regardless of the size of the write. In general if the write is smaller than a multiple of a clustered page, sequential writes perform better than random writes.

4. RocksDB

RocksDB's memtable is the in-memory component (C_0) from an LSM-tree. Every update is first insterted into a memtable, and when it has reached its capacity a new memtable is created for subsequent inserts and the previous memtable is frozen. The frozen memtable then gets written to the disk into a SSTable. The memtables index structure is a skip list, an extension of a sortted linked list.

The SSTable is the on-disk component of a LSM-tree. It is a block-based table format, and stores key-value pairs. The pairs are sorted and is stored with metadata and idexes. The table row firstly contains the key-value pairs, then the metadata. The metadata blocks are data such as Bloom filters, statistics and compression.

5. RocksDB

In RocksDB compaction is the equivalent of rolling merge in LSM-trees. The process takes two or more SSTables and merges their entries to obtain a total ordering. The merged entries becomes a new immutable SSTable. If there is key duplication, only the latest value is included. RockSDB has three different compaction styles; leveled, universal and FIFO.

6. LSM-trees vs B+-trees

LSM-trees are write optimized B+-trees and can batch writes. For B+-trees an insert costs at least one I/O operation, whereas LSM trees can batch writes from the in memory C_0 to the on disk C_1, which greatly reduced the cost for large volumes of inserts. LSM-trees also utilizes sequential writes, which are much faster than random writes.

7.

For hardware faults, this can be hard disks crashing, faulty ram, a power blackout or someone plugs out the wrong cable. Hardware parts have a limited lifetime, for example HDDs. In a storage cluster with 10,000 disks, on average one disk dies per day.

Software faults can be a bug which crashes an application, a process which uses up all the shared resources, a process becomes unresponsive or cascading failures.

Human errors can be a large array of fault. Humans design, built and operate processes, and at any stage something can go wrong. Humans are unreliable by nature. A study around large internet services found out that around 10-25% of all failures were hardware faults, but the rest were human operation.

8.

For hardware faults we can add redundancy and recovery. For redundancy we can have disks running in a RAID configuration, have hot-swappable CPUs, dual power supplies or backup generators in case the power goes out. Since datacenters are growing bigger and bigger, there are more hardware faults and redundancy is not always the correct way. We can tolerate loosing entire machines if we have the proper recovery techniques to get the processes running on other machines.

For software faults there are no recipes or quick solution. One simple way is to simply let processes crash an restart on their own. We can also monitor and analyze systems to find faults quickly. We can also thoroughly test the software to prevent faults.

Lastly human errors we can create well designed APIs and admin interfaces, which creates and abstraction layer, and minimizes opportunities for errors. We can create separate sandbox environments from production environments for testing and developing to decouple production from the place they make the most mistakes. Lastly we can implement good management and training, to both prevent errors and to quickly recover from errors.

9.

The SQL or relational model uses tables and relations between those tables for modeling data. While the document model uses a document format often supported by JSON to store data. This means that document models have more schema flexibility than a SQL model. Also since we can store most data in the same document we get a better performance since we only have to fetch the one document instead of advanced relational queries and joins. One other advantage of the document model is that it better represents data in application code. Much of the application code written today have more similarities with a document model. Using a SQL model application often have to have a layer between such as an ORM to translate the data into the code. However a SQL model is much better at representing relationships between data, such as with the many-to-one and many-to-many relationships.

To model the example using a document model:

Firstly each author is its own document where it stores a list of document ids to its papers

```
{
 "name": "John Doe",
 "address": "Address 100",
 "papers": [
    "paperid_1",
    "paperid_2",
    "paperid_3"
 ]
}
```

Finally a paper can be represented as a document with relations to the authors

```
{
    "sections": [
        {"heading": "This is a heading", "content":"this is the section
        ↪  content"},
        {"heading": "This is a heading", "content":"this is the section
        ↪  content"},
        {"heading": "This is a heading", "content":"this is the section
        ↪  content"},
        {"heading": "This is a heading", "content":"this is the section content"}
    ],
    "authors": [
        "authorid_1",
        "authorid_2"
    ]
}
```

One of the biggest problems is that we cant query the document model with joins as the SQL model, we have to fetch the paper and then fetch each author to get all the data.

10.

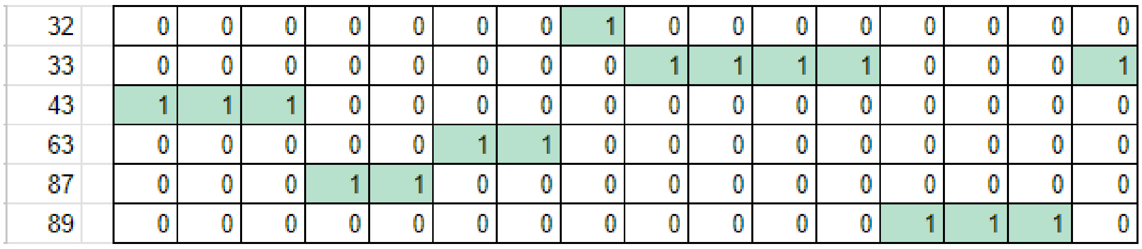
If the many-to-many relationship is common in your data, a graph model would be preferred. For example if you try to model a social network with a document model, the friends list would be long and complex. However in a graph model a vertex could represent a person and each edge could represent a friendship. A benefit of using a graph model is that we can employ well known graph algorithms. It is also a more visual way to look at data, as they are related to each other using edges.

11.

Since textual encodings are human readable, it is often preferred when you encode data for sending to outside organizations. This is because it is hard for everyone to use the same encoding format as is needed with binary encoding.

12. Column Compression

a: bitmap



32	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	1
43	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
63	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
87	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
89	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

Figure 1: Bitmap of column data

b: runlength encoding

- 32: 7,1 (7 zeroes, 1 one, rest zeroes)
- 33: 8,4,3,1 (8 zeroes, 4 ones, 3 zeroes, 1 one)
- 43: 0,3 (0 zeroes, 3 ones, rest zeroes)
- 63: 5,2 (5 zeroes, 2 ones, rest zeroes)
- 87: 3,2 (3 zeroes, 2 ones, rest zeroes)
- 89: 12,3 (12 zeroes, 3 ones, rest zeroes)

13.

MessagePack

MessagePack encodes JSON into a byte sequence. So to add a new field we can simply add add the field to a JSON object:

```
{
 "userName": "Martin",
 "favoriteNumber": 1337,
 "interests": ["daydreaming", "hacking"],
 "labourUnion": "abc123"
}
```

Since MessagePack simply translates the JSON data into a binary encoding it does not include any form of backwards or forward combability. The byte sequences contains values and information about the field. and If we add a field in the Person struct, and old decoder will not be able to put that data into a struct. Likewise if we delete a field, a new decoder wont be able to place old records into the new struct,

Apache Thrift

To add a field to the Person structure with Apache Thrift we can add a field to the schema using the Thrift interface definition language (IDL). Which can be seen under.

```
struct Person {
 1: required string userName,
 2: optional i64 favoriteNumber,
 3: optional list<string> interests,
 4: optional string labourUnion
}
```

Thrift then takes the schema definition and generates code classes for different programming languages. Applications can then use the classes to encode and decode records.

Since Thrift uses field tags it is crucial that all fields have unique field tags. You can change a fields name, but not its tag. Apache Thrift supports forwards compatibility because old code can read new records. This is because if the code reconizes a field tag is does not know about it can use the field type to simply skip that field. It also supports backwards compatibility with a catch. Each new field must either be optional or have a default value. If new code read a old record which did not have the new field, it would fail.

Protocol Buffers

To add a field to the Person structure in Protocol Buffers we can add a field to the schema.

```
message Person {
 required string user_name = 1;
 optional int64 favorite_number = 2;
 repeated string interests = 3;
 optional string labour_union = 4;
}
```

Protocol buffers also has the same code generation as Apache Thrift. It also supports the same forward and backwards compability as Apache Thrift.

Avro

Avro also uses schemas, but has a human readable schema language, Avro IDL and a more machine readable language based on JSON. To add the labour union field we can simply add it to the schema using Avro IDL.

```
record Person {
 string userName;
 union { null, long } favoriteNumber = null;
 array<string> interests;
 union {null, string} labourunion = null;
}
```

One thing to notice here is that there in no tag numbers as in Thrift and Protocol Buffers.

Avro has two types of schemas, a writer's schema and a reader's schema. The writers schema is used when encoding records, and readers schema is used for decoding schemas, the two schemas does not have to have the same version. This is possible because Avro resolves any differences between them on decoding.

With Avro, forward compatibility is meant by that an application has a new version of the writer's schema and another application can have and old version of the reader's schema. For backwards compatibility this is simply in reverse.

To sustain backwards and forward compatibility, all new or deleted fields needs to have a default value. When the new reader's schema decodes a message is simply uses the default value to fill in for missing fields.