A10 + A11

Type	Homework

Bloom Filter

1. **Bloom filters may return a false positive. What is a false positive and how can a false positive happen?**

   false positive is when a bloom filter tells us that the key is in the sstable even though it is not, false positives happen because of collisions.

2. **Bloom filters cannot be used for range queries. Explain why.**

   Suppose you want to check:

   > Is there any element in the range [15, 20]?

   A Bloom filter:

   - Has **no knowledge of ordering or numerical relationships** between elements.
   - Only works by hashing **individual, exact values**.
   - Cannot interpret or reconstruct whether `x ∈ [15, 20]` based on its bit pattern.

   ### Example:

   If you stored the numbers `{3, 7, 19}` in a Bloom filter, and queried for:

   - `is 19 in the set?` → maybe
   - `is 18 in the set?` → either "maybe" or "definitely not"

   But asking:

   - `is there any value in [15, 20]?` → can't be answered without individually checking **each** number in the range (which defeats the purpose).

LSM Storage

h1(1) = 1 mod 20 = 1

h2(1) = 3 mod 20 = 3

h3(1) = 7 mod 20 = 7

h1(12) = 12 mod 20 = 12

h2(12) = 36 mod 20 = 16

h3(12) = 84 mod 20 = 4

h1(7) = 7 mod 20 = 7

h2(7) = 21 mod 20 = 1

h3(7) = 49 mod 20 = 9

h1(18) = 18 mod 20 = 18

h2(18) = 54 mod 20 = 14

h3(18) = 126 mod 20 = 6

After store:

B2 = 0000 0010 0000 0010 0010

h1(8) = 8 mod 20 = 8

h2(8) = 24 mod 20 = 4

h3(8) = 56 mod 20 = 16

After store:

B2 = 0000 1010 1000 0010 1010

h1(12) = 12 mod 20 = 12

h2(12) = 36 mod 20 = 16

h3(12) = 84 mod 20 = 4

After Delete (treat it like store):

B2 = 0000 1010 1010 0010 1010

LSM Storage

Task 2: Write down the content of segment 1 and segment 2 (keys and values) as stored on disk.

Segment 1:

1: "Mariam", 7: "Rudi", 12: “Otto”

Segment 2:

8: "Guga", 12: TOMBSTONE, 18: "Ana"

LSM Storage

Task 3: Process the following read requests:

Segment 1: **B1 = 0101 1001 0100 1000 1000**, Segment 2: **B2 = 0000 1010 1000 0010 1010**

1. **get(key: 3)**

   In bloom filter for segment 2:

   h1(3) = 3 mod 20 = 3 → 0

   h2(3) = 9 mod 20 = 9 → 0

   h3(3) = 21 mod 20 = 1 → 0

   so definitely not in the segment 2, so we got to the bloom filter for the segment 1:

   h1(3) = 3 mod 20 = 3 → 1

   h2(3) = 9 mod 20 = 9 → 1

   h3(3) = 21 mod 20 = 1 → 1

   so it might be in the segment 1, we go to the segment 1 and find that the key 3 is not there so we got a false positive, because of collisions.

2. **get(key: 12)**

   we check the bloom filter for the segment 2:

   h1(12) = 12 mod 20 = 12 → 1

   h2(12) = 36 mod 20 = 16 → 1

   h3(12) = 84 mod 20 = 4 → 1

   so the key might be in the segment 2, so we go to segment 2 and get the value tombstone.

3. **get(key: 14)**

   we check the bloom filter for the segment 2:

   h1(14) = 14 mod 20 = 14 → 1

   h2(14) = 42 mod 20 = 2 → 0

   h3(14) = 98 mod 20 = 18 → 1

   the key is definetly not in the segment 2, so we skip this segment and go to the bloom filter for the segment 1:

   h1(14) = 14 mod 20 = 14 → 0

h2(14) = 42 mod 20 = 2 → 0

h3(14) = 98 mod 20 = 18 → 0

so the key is not in the segment 1 either, key is not in the database.

LSM Storage

Task 4: Merge segments 1 and 2. Assume that the merged segment can hold more than 3 keys. Assume that segment 1 and 2 are the only segments. Build the new bloom filter.

Segment 1:

1: "Mariam", 7: "Rudi", 12: “Otto”

Segment 2:

8: "Guga", 12: TOMBSTONE, 18: "Ana"

Merged Segment 3:

1: “Mariam”, 7: “Rudi”, 8: “Guga”, 18: “Ana”.

1. h1(1) = 1 mod 20 = 1
   h2(1) = 3 mod 20 = 3
   h3(1) = 7 mod 20 = 7
   After store:
   B3 = 0101 0001 0000 0000 0000

2. h1(7) = 7 mod 20 = 7
   h2(7) = 21 mod 20 = 1
   h3(7) = 49 mod 20 = 9
   After store:
   B3 = 0101 0001 0100 0000 0000

3. h1(8) = 8 mod 20 = 8
   h2(8) = 24 mod 20 = 4
   h3(8) = 56 mod 20 = 16
   After store:
   B3 = 0101 1001 1100 0000 1000

4. h1(18) = 18 mod 20 = 18
   h2(18) = 54 mod 20 = 14
   h3(18) = 126 mod 20 = 6
   After store:
   B3 = 0101 1011 1100 0010 1010

Postgres Bloom Filter

Task: Get the Postgres Bloom Filter to work and evaluate it

Activate the Postgres extension and verify:

```
Create extension bloom;
SELECT * FROM pg_extension WHERE extname = 'bloom';
```

Result:

```
agency=#  Create extension bloom; SELECT * FROM pg_extension WHERE extname = 'bloom';
CREATE EXTENSION
  oid  | extname | extowner | extnamespace | extrelocatable | extversion | extconfig | extcondition
-------+---------+----------+--------------+----------------+------------+-----------+--------------
 16598 | bloom   |       10 |         2200 | t              | 1.0        |           |
(1 row)
```

Column	Meaning
`oid`	Internal object ID of the extension (`16598` here).
`extname`	Name of the extension (`bloom`).
`extowner`	Owner's role ID (user ID 10 = default superuser).
`extnamespace`	Namespace ID for where it’s stored (`2200` = usually `public`).
`extrelocatable`	`t` means the extension can be moved between schemas.
`extversion`	Version of the extension (`1.0`).
`extconfig`	Configuration tables used (empty here).
`extcondition`	Any conditions needed for installation (empty = no special conditions).

Postgres Bloom Filter

Create a test table wanted:

```
CREATE TABLE IF NOT EXISTS public.wanted
(
    id serial,
    f_name text COLLATE pg_catalog."default",
    l_name text COLLATE pg_catalog."default",
    city text COLLATE pg_catalog."default",
    birthday text COLLATE pg_catalog."default",
    passport_number text COLLATE pg_catalog."default",
    issuing_country text COLLATE pg_catalog."default",
    CONSTRAINT wanted_pkey PRIMARY KEY (id)
)
TABLESPACE pg_default;

ALTER TABLE IF EXISTS public.wanted
OWNER to postgres;
```

Apart from the PK, all columns are of type 'text'. Explain why.

because, bloom extension requires indexing on multiple columns to create a Bloom filter-based index. text is a supported type for Bloom indexes. Keeping all relevant columns as text ensures they can be uniformly and easily added to the Bloom index without type mismatch issues.

Postgres Bloom Filter

Populate table wanted with 100 000 to 200 000 rows. Have the content somehow generated, for example:

```
INSERT INTO wanted(f_name, l_name, city, birthday, passport_number, issuing_country)
SELECT
  'First' || trunc(random()*1000)::int,
  'Last' || trunc(random()*1000)::int,
  'City' || trunc(random()*500)::int,
  to_char(date '1950-01-01' + trunc(random()*25000)::int, 'YYYY-MM-DD'),
  'P' || lpad(trunc(random()*10000000)::int::text, 8, '0'),
  CASE WHEN random() < 0.5 THEN 'US' ELSE 'DE' END
FROM generate_series(1, 200000);
```

Run Analyze to update statistics

```
ANALYZE wanted;
```

(`ANALYZE` collects statistics about the contents of tables in the database, and stores the results in the `pg_statistic` system catalog. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.)

```
SELECT * FROM pg_stats WHERE tablename = 'wanted';
```

Result:

```
agency=#         SELECT * FROM pg_stats WHERE tablename = 'wanted';
-[ RECORD 1 ]----------+------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
schemaname             | public
tablename              | wanted
attname                | issuing_country
inherited              | f
null_frac              | 0
avg_width              | 3
n_distinct             | 2
most_common_vals       | {DE,US}
most_common_freqs      | {0.50126666,0.49873334}
histogram_bounds       |
correlation            | 0.50125897
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
range_length_histogram |
range_empty_frac       |
range_bounds_histogram |
-[ RECORD 2 ]----------+------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
schemaname             | public
tablename              | wanted
attname                | id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -1
most_common_vals       |
most_common_freqs      |
```

Run a query: Query for any passport_number that is not in the table. Explain the query execution.

```
EXPLAIN SELECT passport_number FROM wanted WHERE passport_number = 'P15068892';
```

Result:

```
agency=# EXPLAIN SELECT passport_number FROM wanted WHERE passport_number = 'P15068892';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on wanted  (cost=0.00..4562.00 rows=1 width=10)
   Filter: (passport_number = 'P15068892'::text)
(2 rows)
```

Seq Scan on wanted: PostgreSQL is performing a **sequential scan**, which means it is checking every row in the `wanted` table.

cost=0.00..4562.00: This is the estimated cost of the query, where:

- `0.00` is the startup cost.
- `4562.00` is the total cost to read the whole table.

rows=1: PostgreSQL estimates that only 1 row matches the condition (The line `rows=1` in the EXPLAIN output is **not the actual number of matching rows**, but rather **PostgreSQL's estimate** based on its statistics.).

width=10: The average size (in bytes) of the `passport_number` column.

Filter: The condition applied to each row: `passport_number = 'P15068892'`.

Postgres Bloom Filter

1. **Create a bloom filter on 6 columns. Use default parameter settings.**

   Enable the Bloom extension (run once per database)

   ```
   CREATE EXTENSION IF NOT EXISTS bloom;
   ```

   Create a Bloom index on the 6 columns with default parameters:

   ```
   CREATE INDEX wanted_bloom_idx ON wanted
   USING bloom (f_name, l_name, city, birthday, passport_number, issuing_country);
   ```

   By default, each column gets 2 bits (`length = 80` bits total) unless you override it with `WITH (colname = N)` options.

2. **Verify that the bloom filter was created.**

   ```
   SELECT indexname, indexdef FROM pg_indexes WHERE tablename = 'wanted';
   ```

   Result:

   ```
   agency=# CREATE INDEX wanted_bloom_idx ON wanted USING bloom (f_name, l_name, city, birthday, passport_number,issuing_country);
   CREATE INDEX
   agency=# SELECT indexname, indexdef FROM pg_indexes WHERE tablename = 'wanted';
       indexname     |                                                  indexdef
   ------------------+------------------------------------------------------------------------------------------------------------------------
    wanted_pkey      | CREATE UNIQUE INDEX wanted_pkey ON public.wanted USING btree (id)
    wanted_bloom_idx | CREATE INDEX wanted_bloom_idx ON public.wanted USING bloom (f_name, l_name, city, birthday, passport_number, issuing_country)
   (2 rows)
   ```

3. **Explain:**

   a. **how many bit arrays will be created and filled?**

      One bit array per row in the table.

      Since we have **200,000 rows**, **200,000 bit arrays** will be created and filled — one per row stored in the bloom index.

      Each bit array corresponds to a single index entry.

   b. **length of the bit arrays?**

      80 bits by default.

      This is controlled by the parameter `length`, which defaults to `80` (can be customized via `WITH (length = N)` when creating the index).

      So each bit array is 80 bits long.

   c. **Which "elements" will be hashed into one bit array?**
      6, The
      values of all indexed columns in that row. In our case we have 6 columns, so each of these 6 column values from a **single row** are hashed and their bits are OR-ed together into that row's **single 80-bit array**.

   d. **How many hash functions are used?**

      2 hash functions per column, by default. So for 6 columns we will have 2×6=12, 12 hash functions used.

If we want to change default settings:

```
CREATE INDEX wanted_bloom_idx ON wanted
USING bloom (
   f_name,
   l_name,
   city,
   birthday,
   passport_number,
   issuing_country
)
WITH (
      --Each row's Bloom filter (bit array) will be 128 bits long instead of the default 80 bits.
      length = 128,
   f_name = 4, -- used 4 hash functions instead of default 2, for this column
   l_name = 3,
   city = 2,
   birthday = 2,
   passport_number = 3,
   issuing_country = 2
);
```

4. **Run the query on passport_number again and explain its execution.**

   ```
   EXPLAIN SELECT passport_number FROM wanted WHERE passport_number = 'P15068892';
   ```

   Result:

```
agency=# EXPLAIN SELECT passport_number FROM wanted WHERE passport_number = 'P15068892';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on wanted  (cost=3076.00..3080.01 rows=1 width=10)
   Recheck Cond: (passport_number = 'P15068892'::text)
   ->  Bitmap Index Scan on wanted_bloom_idx  (cost=0.00..3076.00 rows=1 width=0)
         Index Cond: (passport_number = 'P15068892'::text)
(4 rows)
```

   Before:

   - PostgreSQL did a **sequential scan**: it checked **every row** in the `wanted` table (all 200,000 rows).
   - It had **no index** to assist in filtering, so performance was linear.
   - Estimated cost: `0.00..4562.00`

   After:

   - PostgreSQL now uses a **Bloom index** (`wanted_bloom_idx`).
   - First it does a **Bitmap Index Scan** using the Bloom filter to get **candidate rows**.
   - Then a **Bitmap Heap Scan** re-checks those candidates on the actual table (`Recheck Cond`).
   - Estimated cost: `0.00..3080.01` (lower index scan cost, more efficient).

5. **Create a btree index on passport_number**

   ```
   CREATE INDEX idx_passport_number_btree ON wanted(passport_number);
   ```

   Unlike the **Bloom index**, B-tree indexes:

   - Do **not need recheck** conditions (they are exact).
   - Are **more efficient for single-column lookups** like this.

6. **Run the query on passport_number again and explain its execution.**

   ```
   EXPLAIN SELECT passport_number FROM wanted WHERE passport_number = 'P15068892';
   ```

```
CREATE INDEX
agency=# EXPLAIN SELECT passport_number FROM wanted WHERE passport_number = 'P15068892';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Only Scan using idx_passport_number_btree on wanted  (cost=0.42..4.44 rows=1 width=10)
   Index Cond: (passport_number = 'P15068892'::text)
(2 rows)
```

- PostgreSQL performs a direct **Index Scan** using the B-tree index.
- It **retrieves only matching rows** (no extra filtering or recheck).
- **No Bitmap scan** or **Recheck Cond**.
- Estimated cost is much lower (e.g. `cost=0.42..4.44` depending on stats).
- **Exact match**, no false positives.

Feature	Bloom Index	B-tree Index
Recheck required?	✅ Yes	❌ No
Supports range queries?	❌ No	✅ Yes
Supports sorting (`ORDER`)	❌ No	✅ Yes
Ideal for	Multi-column filters	Single-column lookups
False positives possible?	✅ Yes	❌ No

Postgres Bloom Filter

Evaluate the Postgres bloom filter:

1. **What main disadvantages do you see?**

Disadvantage	Explanation
False positives	Bloom filters are *lossy*; they may report matches that aren't real. PostgreSQL must re-check the actual table (heap) to confirm matches. This adds I/O.
Equality-only support	Works **only** for `=` (equality) comparisons. No support for range queries (`<` , `>` , `LIKE` , etc.).
Not MVCC-aware	Bloom indexes **don't support index-only scans**. They can't determine visibility — the table must always be consulted.
Manual tuning required	You must manually assign bits per column and length — no automatic optimization like with B-trees.
Fixed size per row	Each row gets the full bit array even if only a few columns are used in a query. Potentially wasteful on storage compared to targeted B-tree indexes.
Less tested and mature	It's a **contrib module**, not core. Fewer production deployments compared to B-tree or GIN/GiST indexes.
Worse for small data	Overhead of hashing and rechecking makes it less beneficial on small tables where a seq scan or B-tree is faster.

2. **What could be use cases for the Postgres bloom filter?**

Use Case	Why Bloom is Good
Wide tables with many filter columns	When you filter by **4–10+ columns using** `=` , Bloom avoids needing a huge composite B-tree.
Queries with sparse results	If few rows match and you use many columns in WHERE clauses, Bloom minimizes heap reads.
Read-heavy analytical workloads	Works well in **OLAP-style** querying where wide filters are common and updates are rare.
Archival/search tables	Tables that are mostly read-only, like **logs, blacklists, or "wanted" lists**. Bloom avoids full scans without needing complex index combos.

Use Case	Why Bloom is Good
Low false-positive tolerance is acceptable	When the cost of verifying false matches is cheaper than full scans or maintaining B-tree indexes.

MongoDB: Basic Queries

1. **How many male teachers are in teacher collection?**

```
db.teacher.countDocuments({ t_gender: "m" })
```

```
> db.teacher.countDocuments({ t_gender: "m" })
< 10
```

2. **Return documents of teachers that live in postalcode 4600. Do not show object_id.**

```
db.teacher.find({ t_postalcode: 4600 }, { _id: 0 })
```

```
> db.teacher.find({ t_postalcode: 4600 }, { _id: 0 })
< {
    t_id: 8,
    t_name: 'Duerr',
    t_mail: 'duerr@galopp.xx',
    t_postalcode: 4600,
    t_dob: '2002-06-01',
    t_gender: 'm',
    t_education: 'HighSchool',
    t_remark: null,
    t_payment: 0
  }
  {
    t_id: 9,
    t_name: 'Schumann',
    t_mail: 'schumann@galopp.xx',
    t_postalcode: 4600,
    t_dob: '1994-10-17',
    t_gender: 'f',
    t_education: 'Bachelor',
    t_remark: null,
    t_payment: 0
  }
```

3. **Return all documents of male teachers in postalcode 4600.**

```
db.teacher.find({ t_gender: "m", t_postalcode: 4600 })
```

```
> db.teacher.find({ t_gender: "m", t_postalcode: 4600 })
< {
    _id: ObjectId('680b6b54514adfb09a1292de'),
    t_id: 8,
    t_name: 'Duerr',
    t_mail: 'duerr@galopp.xx',
    t_postalcode: 4600,
    t_dob: '2002-06-01',
    t_gender: 'm',
    t_education: 'HighSchool',
    t_remark: null,
    t_payment: 0
  }
  {
    _id: ObjectId('680b6b54514adfb09a1292e2'),
    t_id: 12,
    t_name: 'Schmidt',
    t_mail: 'schmidt@immer_da.xx',
    t_postalcode: 4600,
    t_dob: '2003-10-03',
    t_gender: 'm',
    t_education: 'Master',
    t_remark: null,
    t_payment: 0
  }
```

4. **Return the documents of teachers Krawinkel and Kaiser (or condition on any other two teachers)**

```
db.teacher.find({ t_name: { $in: ["Krawinkel", "Kaiser"] } })
```

```
> db.teacher.find({ t_name: { $in: ["Krawinkel", "Kaiser"] } })
< {
    _id: ObjectId('680b6b54514adfb09a1292f0'),
    t_id: 2,
    t_name: 'Kaiser',
    t_mail: 'kaiser@circus.xx',
    t_postalcode: 4500,
    t_dob: '2001-11-30',
    t_gender: 'f',
    t_education: 'HighSchool',
    t_remark: 'The relational model was a theoretical proposal, and many people at the time doubted whether it could be implemented efficien
    t_payment: 6
  }
  {
    _id: ObjectId('680b6b54514adfb09a1292f1'),
    t_id: 1,
    t_name: 'Krawinkel',
    t_mail: 'krawinkel@we-are.xx',
    t_postalcode: 4500,
    t_dob: '1978-09-15',
    t_gender: 'f',
    t_education: 'Master',
    t_remark: 'In a complex application there may be more intermediary levels, such as APIs built upon APIs, but the basic idea is still the
    t_payment: 10
```

5. **Teacher Krawinkel teaches EN, DE and FR. Add the subjects to the document. (updateOne())**

```
db.teacher.updateOne(
  { t_name: "Krawinkel" },
  { $set: { subjects: ["EN", "DE", "FR"] } }
)
```

```
> db.teacher.updateOne(
    { t_name: "Krawinkel" },
    { $set: { subjects: ["EN", "DE", "FR"] } }
  )
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
```

```
> db.teacher.find({t_name: "Krawinkel"})
< {
    _id: ObjectId('680b6b54514adfb09a1292f1'),
    t_id: 1,
    t_name: 'Krawinkel',
    t_mail: 'krawinkel@we-are.xx',
    t_postalcode: 4500,
    t_dob: '1978-09-15',
    t_gender: 'f',
    t_education: 'Master',
    t_remark: 'In a complex application there may be more intermediary levels, such as APIs built upon APIs, but the basi
    t_payment: 10,
    subjects: [
      'EN',
      'DE',
      'FR'
    ]
}
```

6. **Teacher Alt teaches MA, PH and CH. Add the subjects to the document**

```
db.teacher.updateOne(
  { t_name: "Alt" },
  { $set: { subjects: ["MA", "PH", "CH"] } }
)
```

```
> db.teacher.updateOne(
    { t_name: "Alt" },
    { $set: { subjects: ["MA", "PH", "CH"] } }
  )
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
> db.teacher.find({t_name: "Alt"})
< {
    _id: ObjectId('680b6b54514adfb09a1292e7'),
    t_id: 18,
    t_name: 'Alt',
    t_mail: 'alt@neu.xx',
    t_postalcode: 4600,
    t_dob: '1965-05-03',
    t_gender: 'm',
    t_education: 'Master',
    t_remark: null,
    t_payment: 0,
    subjects: [
      'MA',
      'PH',
      'CH'
    ]
  }
```

Setting Validation Rules

Set the following validation rules for teacher collection:

1. **required fiedls: 't_name', 't_mail', 't_gender'.**

```
db.runCommand({
  collMod: "teacher",
  validator: {
    $jsonSchema: {
      required: ["t_name", "t_mail", "t_gender"]
    }
  },
  validationLevel: "strict",
  validationAction: "error"
});
```

2. **birthday: needs to be a date field**

```
db.runCommand({
  collMod: "teacher",
  validator: {
    $jsonSchema: {
      properties: {
        birthday: { bsonType: "date" }
      }
    }
  },
  validationLevel: "strict",
  validationAction: "error"
});
```

3. **gender needs to be an enum field that allows only one of the enum values (one gender!)**

```
db.runCommand({
  collMod: "teacher",
  validator: {
    $jsonSchema: {
      properties: {
        t_gender: {
          bsonType: "string",
          enum: ["Male", "Female", "Other"]
        }
      }
    }
  },
  validationLevel: "strict",
  validationAction: "error"
});
```

4. **education needs to be an enum field with values "High School", "Bachelor" and "Master". Multiple values are allowed.**

```
db.runCommand({
  collMod: "teacher",
  validator: {
    $jsonSchema: {
```

```
      properties: {
        education: {
          bsonType: "array",
          items: {
            bsonType: "string",
            enum: ["High School", "Bachelor", "Master"]
          },
          uniqueItems: true
        }
      }
    }
  },
  validationLevel: "strict",
  validationAction: "error"
});
```

5. **subjects needs to be an enum field with a couple of subjects as values. Multiple values are allowed.**

```
db.runCommand({
  collMod: "teacher",
  validator: {
    $jsonSchema: {
      properties: {
        subjects: {
          bsonType: "array",
          items: {
            bsonType: "string",
            enum: [
                "DE", "EN", "FR"
            ]
          },
          uniqueItems: true
        }
      }
    }
  },
  validationLevel: "strict",
  validationAction: "error"
});
```

ALL TOGETHER:

```
db.runCommand({
  collMod: "teacher",
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["t_name", "t_mail", "t_gender"],
      properties: {
        t_name: { bsonType: "string" },
        t_mail: { bsonType: "string" },
        t_gender: {
          bsonType: "string",
          enum: ["Male", "Female", "Other"]
        },
        birthday: { bsonType: "date" },
```

```
      education: {
        bsonType: "array",
        items: {
          bsonType: "string",
          enum: ["High School", "Bachelor", "Master"]
        },
        uniqueItems: true
      },
      subjects: {
        bsonType: "array",
        items: {
          bsonType: "string",
          enum: [
            "DE", "EN", "FR"
          ]
        },
        uniqueItems: true
      }
    }
  },
  validationLevel: "strict",
  validationAction: "error"
});
```

```
{
   $jsonSchema: {
     bsonType: "object",
     required: ["t_name", "t_mail", "t_gender"],
     properties: {
       t_name: { bsonType: "string" },
       t_mail: { bsonType: "string" },
       t_gender: {
         bsonType: "string",
         enum: ["Male", "Female", "Other"]
       },
       birthday: { bsonType: "date" },
       education: {
         bsonType: "array",
         items: {
           bsonType: "string",
           enum: ["High School", "Bachelor", "Master"]
         },
         uniqueItems: true
       },
       subjects: {
         bsonType: "array",
         items: {
           bsonType: "string",
           enum: [
             "DE", "EN", "FR"
           ]
         },
         uniqueItems: true
       }
     }
```

```
  }
 }
```

Insert a Document in to a Collection, Verify Validation

Insert the following document into the teacher collection. Use a command that inserts exactly one document.:

t_name: “Smith",
t_mail: “jenny.smith@super.xx",
t_subjects: ["DE", "EN", "FR"],
t_education: ["Bachelor“, "Master"],
t_dob: "1985-03-20",
t_remark: "I have 10 years of experience with private teaching. Students trust me and I will make students more successful. I charge very moderate prices.“,
t_gender: “f”

Insert should fail! Why? Correct the error.

```
db.teacher.insertOne({
  t_name: "Smith",
  t_mail: "jenny.smith@super.xx",
  t_subjects: ["DE", "EN", "FR"],
  t_education: ["Bachelor", "Master"],
  t_dob: "1985-03-20",
  t_remark: "I have 10 years of experience with private teaching. Students trust me and I will make students more success
  t_gender: "f"
});
```

The insert failed because of multiple reasons:

Error	Fix
`t_gender: "f"` not valid	Changed to `"Female"`
Wrong field name `t_subjects`	Changed to `subjects`
Wrong field name `t_education`	Changed to `education`
Wrong field name & type `t_dob`	Changed to `birthday: new Date("1985-03-20")`

Insert a Documentinton Collection, Verify Validation

Insert the following document into the teacher collection. Use a command that inserts exactly one document.:

t_name: “Young",
t_mail: “jenny.Young@super.xx",
t_subjects: ["DE", "EN", "FR"], // one of the subjects should NOT be defined.
t_education: ["Bachelor“, "Master"],
t_dob: "1985-03-20",
t_remark: "I have 10 years of experience with private teaching. Students trust me and I will make students more successful. I charge very moderate prices.“,
t_gender: “f”

```
db.teacher.insertOne({
  t_name: "Young",
  t_mail: "jenny.Young@super.xx",
  t_subjects: ["DE", "EN", "FR"],
  t_education: ["Bachelor", "Master"],
  t_dob: "1985-03-20",
  t_remark: "I have 10 years of experience with private teaching. Students trust me and I will make students more success
  t_gender: "f"
});
```

Corrected:

```
db.teacher.insertOne({
  t_name: "Young",
  t_mail: "jenny.Young@super.xx",
  subjects: ["DE", "EN", "FR"],
  education: ["Bachelor", "Master"],
  birthday: new Date("1985-03-20"),
  t_remark: "I have 10 years of experience with private teaching. Students trust me and I will make students more success
  t_gender: "Female"
});
```

Document Update: Add/ Update an Array

1. **The teacher documents imported do not have a subjects' array. Take one of the imported teacher documents, update the document and add a subjects' field.**

   before:

```
> db.teacher.findOne({ _id: ObjectId("680b6b54514adfb09a1292e8") })
< {
    _id: ObjectId('680b6b54514adfb09a1292e8'),
    t_id: 19,
    t_name: 'Engel',
    t_mail: 'engel@galopp.xx',
    t_postalcode: 5400,
    t_dob: '1959-03-15',
    t_gender: 'f',
    t_education: 'HighSchool',
    t_remark: null,
    t_payment: 0
  }
```

multiple things violate the validation so we have to update not only subject, if others did not violate it the code would have looked like this:

```
db.teacher.updateOne(
  { _id: ObjectId("680b6b54514adfb09a1292e8") },
  { $set: { subjects: ["DE", "EN"] } }
);
```

but since they do, the code will look like this:

```
db.teacher.updateOne(
  { _id: ObjectId("680b6b54514adfb09a1292e8") },
  {
    $set: {
      t_gender: "Female",
      birthday: new Date("1959-03-15"),
      education: ["High School"],
      subjects: ["DE", "EN"]
    },
    $unset: {
      t_dob: "",        // remove old invalid birthday field
      t_education: ""   // remove old invalid education field
    }
  }
);
```

After:

```
> db.teacher.findOne({ _id: ObjectId("680b6b54514adfb09a1292e8") })
< {
    _id: ObjectId('680b6b54514adfb09a1292e8'),
    t_id: 19,
    t_name: 'Engel',
    t_mail: 'engel@galopp.xx',
    t_postalcode: 5400,
    t_gender: 'Female',
    t_remark: null,
    t_payment: 0,
    birthday: 1959-03-15T00:00:00.000Z,
    education: [
      'High School'
    ],
    subjects: [
      'DE',
      'EN'
    ]
  }
```

2. **Add another subject to the subjects array. Use function$addToSet and run the same update command twice.**

```
db.teacher.updateOne(
  { _id: ObjectId("680b6b54514adfb09a1292e8") },
  { $addToSet: { subjects: "FR" } }
);

// Run it a second time:
db.teacher.updateOne(
  { _id: ObjectId("680b6b54514adfb09a1292e8") },
  { $addToSet: { subjects: "FR" } }
);
```

running it two times does not change anything the subject is only added once:

`$addToSet` **does NOT allow duplicates**

```
> db.teacher.findOne({ _id: ObjectId("680b6b54514adfb09a1292e8") })
< {
    _id: ObjectId('680b6b54514adfb09a1292e8'),
    t_id: 19,
    t_name: 'Engel',
    t_mail: 'engel@galopp.xx',
    t_postalcode: 5400,
    t_gender: 'Female',
    t_remark: null,
    t_payment: 0,
    birthday: 1959-03-15T00:00:00.000Z,
    education: [
      'High School'
    ],
    subjects: [
      'DE',
      'EN',
      'FR'
    ]
  }
```

3. **Add yet another subject to the subjects array. Use function$push and run the same update command twice.**

```
db.teacher.updateOne(
  { _id: ObjectId("680b6b54514adfb09a1292e8") },
  { $push: { subjects: "GE" } }
);

// Run it a second time:
db.teacher.updateOne(
  { _id: ObjectId("680b6b54514adfb09a1292e8") },
  { $push: { subjects: "GE" } }
);
```

Before we run the code the second time we remove `uniqueItems: true` from validation, then the code will add GE the second time, since `$push` allows duplications.

```
> db.teacher.findOne({ _id: ObjectId("680b6b54514adfb09a1292e8") })
< {
    _id: ObjectId('680b6b54514adfb09a1292e8'),
    t_id: 19,
    t_name: 'Engel',
    t_mail: 'engel@galopp.xx',
    t_postalcode: 5400,
    t_gender: 'Female',
    t_remark: null,
    t_payment: 0,
    birthday: 1959-03-15T00:00:00.000Z,
    education: [
      'High School'
    ],
    subjects: [
      'DE',
      'EN',
      'FR',
      'GE',
      'GE'
    ]
  }
```

4. **Remove one of the duplicate values out of the array.**

```
db.teacher.updateOne(
  { _id: ObjectId("680b6b54514adfb09a1292e8") },
  { $pull: { subjects: "GE" } }
);
```

pull removes all occurances of “GE”, and then manually add one occurance of “GE”.

```
db.teacher.updateOne(
  { _id: ObjectId("680b6b54514adfb09a1292e8") },
  { $push: { subjects: "GE" } }
);
```

Result:

```
> db.teacher.findOne({ _id: ObjectId("680b6b54514adfb09a1292e8") })
< {
    _id: ObjectId('680b6b54514adfb09a1292e8'),
    t_id: 19,
    t_name: 'Engel',
    t_mail: 'engel@galopp.xx',
    t_postalcode: 5400,
    t_gender: 'Female',
    t_remark: null,
    t_payment: 0,
    birthday: 1959-03-15T00:00:00.000Z,
    education: [
      'High School'
    ],
    subjects: [
      'DE',
      'EN',
      'FR',
      'GE'
    ]
  }
```

Insert Embedded Documents

1. **Choose one teacher and add an array of embedded documents holding the students that the teacher teaches:**

Rose Singer, 2014-03-28, 2021-01-15
Donald Black, 2011-05-05, 2020-01-15
Donald Blue, 2017-05-05, 2020-01-15

```
db.teacher.updateOne(
 { _id: ObjectId("680b6b54514adfb09a1292e8") },
 {
  $set: {
   students: [
    {
     name: "Rose Singer",
     birthdate: new Date("2014-03-28"),
     startdate: new Date("2021-01-15")
    },
    {
     name: "Donald Black",
     birthdate: new Date("2011-05-05"),
     startdate: new Date("2020-01-15")
    },
    {
     name: "Donald Blue",
     birthdate: new Date("2017-05-05"),
     startdate: new Date("2020-01-15")
    }
   ]
  }
 }
);
```

Result:

2. **Run a query that searches the collection teacher for the student Donald.**

```
db.teacher.find({
  students: {
    $elemMatch: {
      name: { $regex: "^Donald", $options: "i" }
    }
  }
});
```

Result:

Create a reference between documents

- **Import the student collection**

```
db.student.insertMany([
  {
    name: "Rose Singer",
    birthdate: new Date("2014-03-28"),
    startdate: new Date("2021-01-15")
  },
  {
    name: "Donald Black",
    birthdate: new Date("2011-05-05"),
    startdate: new Date("2020-01-15")
  }
]);
```

- **Create a reference from a student in the student table to two teachers.**

```
db.student.updateOne(
  { name: "Rose Singer" },
  {
    $set: {
      teacher_ids: ["680b6b54514adfb09a1292e8", "680b6b54514adfb09a1292de"]
    }
  }
);
```

- **Query the reference to find the teachers of the student in the teacher table.**

```
db.student.aggregate([
  { $match: { name: "Rose Singer" } },
  {
    $lookup: {
      from: "teacher",
      localField: "teacher_ids",
      foreignField: "_id",
      as: "teachers"
    }
  }
]);
```

```
> db.student.aggregate([
    { $match: { name: "Rose Singer" } },
    {
      $lookup: {
        from: "teacher",
        localField: "teacher_ids",
        foreignField: "_id",
        as: "teachers"
      }
    }
  ]);
< {
    _id: ObjectId('6822658dc30e3c75629ebbc0'),
    name: 'Rose Singer',
    birthdate: 2014-03-28T00:00:00.000Z,
    startdate: 2021-01-15T00:00:00.000Z,
    teacher_ids: [
      '680b6b54514adfb09a1292e8',
      '680b6b54514adfb09a1292de'
    ],
    teachers: []
  }
lesson>
```