Data Modelling & SQL - Day 3

Indexes

Warning

Most of following slides refer to Postgres, though general concepts presented are (partially) applicable to other RDBMS.

Introduction

As already mentioned previously, today's world is overwhelmed with data.

If you would need to find some important piece of information, and you know where to find it (e.g. your notebook) and the storage volume is not large (e.g. 100 pages), then you could try to browse through the pages one-by-one and find the necessary part relatively fast within small amount of time.

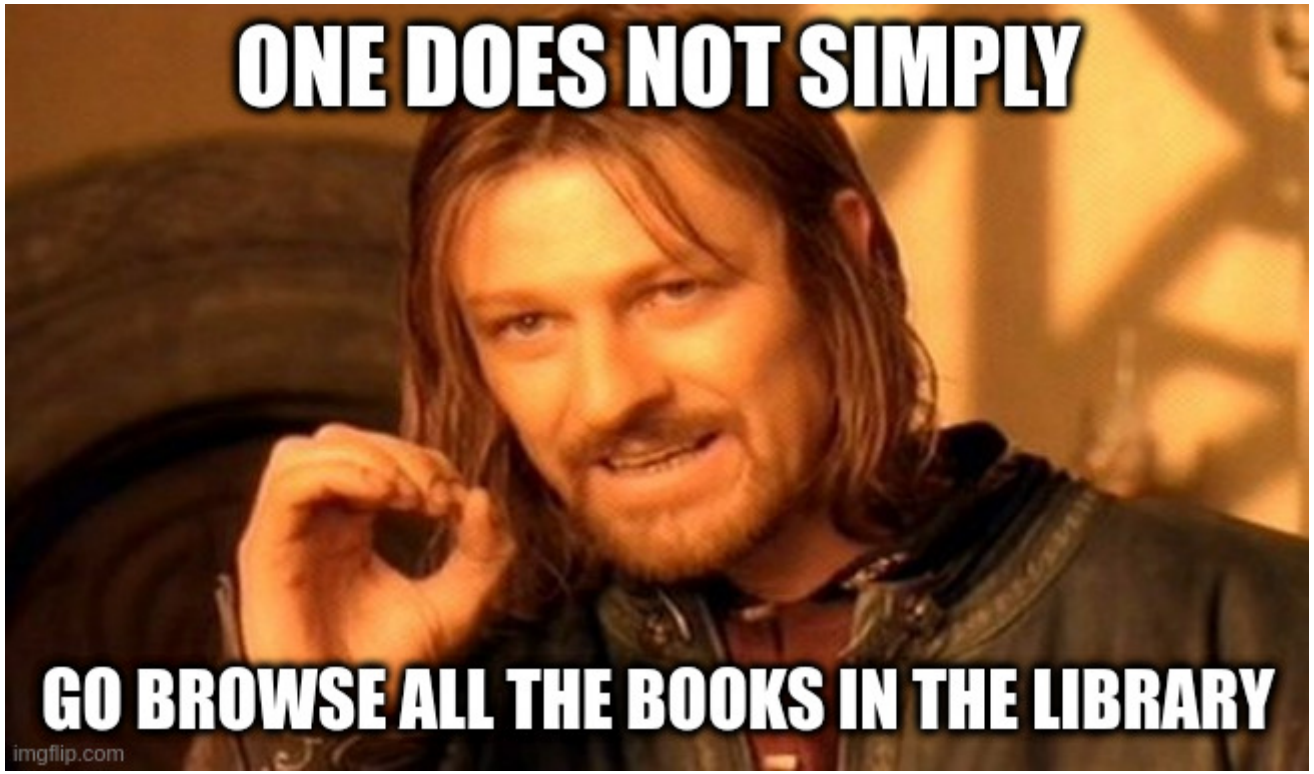
Finding data in a bigger dataset

But what if you don't know where to search (e.g. you have multiple computers). Or you don't know where to look?

If you've already used a search engine (either online, e.g. Google or DuckDuckGo; or offline, e.g. file search on your PC or text search in a document/ multiple documents), you already know that computers are capable of finding the necessary information pretty fast.

Searching in a library

If you're a bit older and remember what a **real** library is, you'd remember, that one doesn't simply go through all the books to find the ones by an author **X**.



Searching by index

Instead, you'd use an index and either go to a section, dedicated to 'Databases' or find the books by a certain author (whose name starts with letters **Geh** (Database Management Systems by Raghu Ramakrishnan, Johannes Gehrke)) or both.

Why should a database be any different and spend unnecessary CPU cycles? Why would we as users want to spend more time waiting for query evaluation result?

Exercise: Guess the number

I’ll write down a number between 0 and 10 and you can all try to guess it.

Let’s see how many tries you need.

Exercise 2: A bit more information

Let’s try again, but this time I’ll tell you if you guessed too low, to high or correctly.

Does this make a difference?

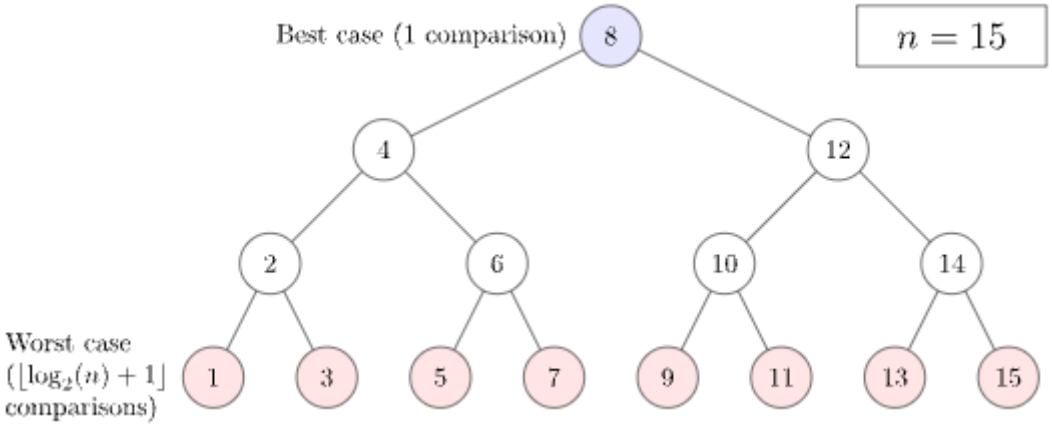
Exercise 3: A bigger dataset

Same exercise, but now with a number between 0 and 1000.

Binary search

- This is an example of a binary search.
- It requires a sorted list.
- Keep on dividing in half multiple times and you quickly find the answer.
- This is how a basic index works.

Binary search illustrated



Binary search performance

# of items	Linear tries	Binary tries
1000	500	10
1000 000	500 000	20
1000 000 000	500 000 000	30
1000 000 000 000	500 000 000 000	40

Demo

Docker & Docker Compose

See `docker-compose.yaml`

```
docker-compose up
```

PgAdmin

Credentials - see `docker-compose.yaml`.

Creating server (connection) to postgres - see `docker-compose.yaml`, use `postgres` (service name) as hostname, `5432` as port (standard).

EXPLAIN ANALYZE

Note: primary key acts as a kind of index.

```
---
--- demo1: usage of explain, influence of adding index
---
create table demo1 (
    key serial primary key,
    value varchar(20)
);

with symbols(characters) as (VALUES ('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 '))
insert into demo1(value) (

    select string_agg(substr(characters, (random() * length(characters) + 1) :: INTEGER, 1), '')
    from symbols
    join generate_series(1,20) as word(chr_idx) on 1 = 1 -- word length
    join generate_series(1,10000) as words(idx) on 1 = 1 -- # of words
    group by idx
);

--- use explain for following queries

--- key supports equality index lookup
select * from demo1 where key = 10;
```

```
--- key supports range index lookup
select * from demo1 where key < 10;

--- value requires sequential table scan (searching for all values starting with <space>)
select * from demo1 where value like ' %';

--- add index
create index idx_demo1_value on demo1(value);

--- value requires sequential table scan, since not on c-locale, see
https://www.postgresql.org/docs/13/indexes-types.html
select * from demo1 where value like ' %';

-- drop index
drop index if exists idx_demo1_value;

--- add index
create index idx_demo1_value on demo1(value text_pattern_ops);

--- value supports index scan now and should be faster
select * from demo1 where value like ' %';
```

Uniqueness

- non-unique (default)
- unique
    - Not always applicable
    - Can be faster

    - Enforces uniqueness

Common index types

- b-tree (default)
    - O(log n)
    - can also handle ranges
- hash
    - O(1) efficiency
    - can only handle equality

Advanced index types

- gin ("inverted indexes", test presence of an element in e.g. array)
- gist (e.g. index support for two-dimensional data structures)
- brin
- geo

Advantages

- Speed-up select queries.
    - Also the WHERE part in updates and deletes.
- Even bigger differences when joining.

- Indexes are easier to store in memory. Less disk I/O.

Disadvantages

- When writing, the index also needs to be updated, costs performance.
    - Often this is not a real issue.
- Order-error-prone.
- Increase storage usage.

Multicolumn indexes

```
---
--- demo2, multicolumn indexes, pitfalls, index size
---
create table demo2 (
    key serial primary key,
    value1 varchar(20),
    value2 varchar(200),
    value3 varchar(2000)
);

insert into demo2(value1, value2, value3) (
    select substr(text, 0, 20), substr(text, 20, 200), substr(text, 220, 2000) from (
        with symbols(characters) as (VALUES ('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
'))
        select string_agg(substr(characters, (random() * length(characters) + 1) :: INTEGER, 1), '') as text
        from symbols
        join generate_series(1,20+200+2000) as word(chr_idx) on 1 = 1 -- word length
        join generate_series(1,10000) as words(idx) on 1 = 1 -- # of words
        group by idx
```

```
    ) texts
);

-- Count the number of inserted rows
select count(*) from demo2;

-- View the generated data
select * from demo2 limit 100;

--- create index on value1
create index idx_demo2_value1 on demo2(value1 text_pattern_ops);

--- create index on value3
create index idx_demo2_value3 on demo2(value3 text_pattern_ops);

--- create index on value1+value3
create index idx_demo2_value1_value3 on demo2(value1 text_pattern_ops, value3 text_pattern_ops);


--- check relation size
select pg_size_pretty(pg_table_size('demo2')) as table_size, pg_size_pretty(pg_total_relation_size('demo2'))
as total_relation_size;
select pg_size_pretty(pg_indexes_size('demo2')) as indexes_size;
select pg_size_pretty(pg_table_size('idx_demo2_value1')) as index1_size,
pg_size_pretty(pg_table_size('idx_demo2_value3')) as index2_size,
pg_size_pretty(pg_table_size('idx_demo2_value1_value3')) as index3_size;

--- use explain for following queries

--- uses idx_demo2_value1
select * from demo2 where value1 like 'a%';

--- still uses idx_demo2_value1, since where starts with 'value1'
select * from demo2 where value1 like 'a%' and value2 like 'b%';
```

```
--- uses idx_demo2_value1_value3
select * from demo2 where value1 like 'a%' and value3 like 'c%';

--- full table scan!, since idx_demo2_value1_value3 starts with 'value2'
select * from demo2 where value2 like 'b%' and value3 like 'c%';
```

Unique indexes

```
---
--- demo3, unique index
---
create table demo3 (
    key serial primary key,
    value varchar(20)
);

--- create index on value1
create index idx_demo3_value on demo3(value text_pattern_ops);

--- insert
insert into demo3 (value) values ('val1');

insert into demo3 (value) values ('val2');

insert into demo3 (value) values ('val1');

--- drop old index
drop index idx_demo3_value;

--- try to add unique index --> error, duplicate value=val1
create unique index udx_demo3_value on demo3(value text_pattern_ops);

--- delete dups
```

```
delete from demo3 where value = 'val1';

--- re-create index as unique
create unique index udx_demo3_value on demo3(value text_pattern_ops);

--- insert
insert into demo3 (value) values ('val1');

--- insert again --> error, duplicate value=val1
insert into demo3 (value) values ('val1');
```

Indexes on expressions

```
---
--- demo4: index on expressions
---
create table demo4 (
    key serial primary key,
    value varchar(20)
);

with symbols(characters) as (VALUES ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'))
insert into demo4(value) (

    select string_agg(substr(characters, (random() * length(characters) + 1) :: INTEGER, 1), '')
    from symbols
    join generate_series(1,20) as word(chr_idx) on 1 = 1 -- word length
    join generate_series(1,400000) as words(idx) on 1 = 1 -- # of words
    group by idx
);

--- create index on lower-case value
create index idx_demo4_value_lc on demo4(lower(value));
```

```
select count(*) from demo4;

--- query uses sequential scan
select * from demo4 where value = 'P';

--- query uses sequential scan, since postgres does not have a crystal ball or AI built-in ;)
select * from demo4 where value = 'p';

--- query uses index scan
select * from demo4 where lower(value) = 'p'
```

Partial indexes

```
---
--- demo5: partial index
---
create table demo5 (
    key serial primary key,
    value varchar(20)
);

with symbols(characters) as (VALUES ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'))
insert into demo5(value) (

    select string_agg(substr(characters, (random() * length(characters) + 1) :: INTEGER, 1), '')
    from symbols
    join generate_series(1,20) as word(chr_idx) on 1 = 1 -- word length
    join generate_series(1,10000) as words(idx) on 1 = 1 -- # of words
    group by idx
);

--- create index on values starting with 'A'
```

```
create index idx_demo5_value_A on demo5(value text_pattern_ops) where value like 'A%';

--- query uses sequential scan
select * from demo5 where value like 'B%';

--- query uses index scan
select * from demo5 where value like 'A%';

--- query uses sequential scan
select * from demo5 where value like 'AA%';

--- query uses sequential scan
select * from demo5 where value = 'AAA';

--- query uses index and then sequential scan
select * from demo5 where value like 'A%' and value like 'AA%';

--- query uses index and then sequential scan
select * from demo5 where value like 'A%' and value = 'AAA';
```

Transactions

Introduction

- Until now, we’ve seen how we can insert/update/delete data.
- But these were single operations.
- Let’s say we’d like to do multiple operations in sequence…

An example: Banking

- Let’s say we have a banking system and database.
- We have a table "Accounts".
- Every account has a "balance" column.

An example: Banking

- Let’s move some money between accounts
- UPDATE accounts SET balance = balance-100 WHERE id='1';
  UPDATE accounts SET balance = balance+100 WHERE id='2';
- Simple, right?

An example: Banking

- But what if the second update fails?
- Simple, just fix it in a third update:
  UPDATE accounts SET balance = balance+100 WHERE id='1';
- But what if this one also fails?
- And do we really want to clutter our code with this error handling?
- Also, there’s a lot more can go wrong…

Transactions

- Transactions are the tool for this.
- Simply put, instead of running 2 separate queries, you can run them as part of one transaction.
- A transaction which succeeds or aborts.
- Succeeding is called a "commit".
- Failing is called a "rollback".

An example with transactions

- Again the banking example
- BEGIN TRANSACTION;
  UPDATE accounts SET balance = balance-100 WHERE id='1';
  UPDATE accounts SET balance = balance+100 WHERE id='2';
  COMMIT;
- Simple, right?

Exercise: Transactions

- Let’s update the banking example to let it work with a transaction.
- First create an "accounts" table, with at least a column "balance".
- Insert some rows to get started.
- Then, in a transaction, send some money from one account to the second account.

Exercise: Solution 1/3

```
CREATE TABLE accounts(
    id serial not null primary key,
    balance int not null
);
```

Exercise: Solution 2/3

```
INSERT INTO accounts(balance) VALUES
(100),
(100)
```

Exercise: Solution 3/3

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance-10 WHERE id=1;
UPDATE accounts SET balance = balance+10 WHERE id=2;
COMMIT;
```

ACID

- When searching for transactions, you'll often see the term ACID.
- ACID stands for
    - Atomicity

- Consistency
- Isolation
- Durability

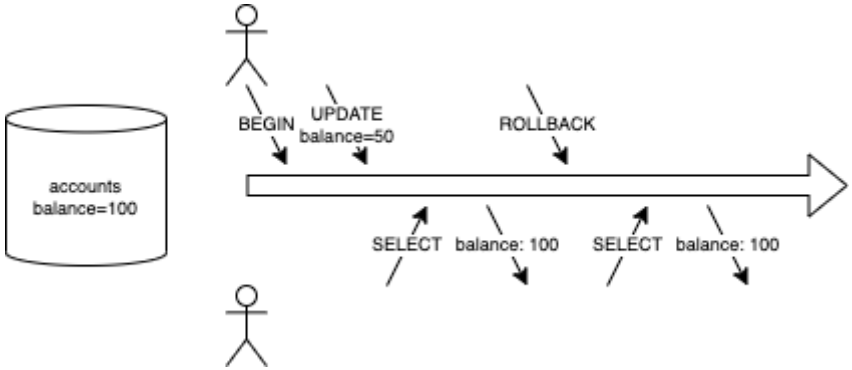
ACID meaning

- ACID are properties that transactions provide.
- No error, hardware failure or whatever may impact these.
- So, the ACID properties let you write safer software using transactions.

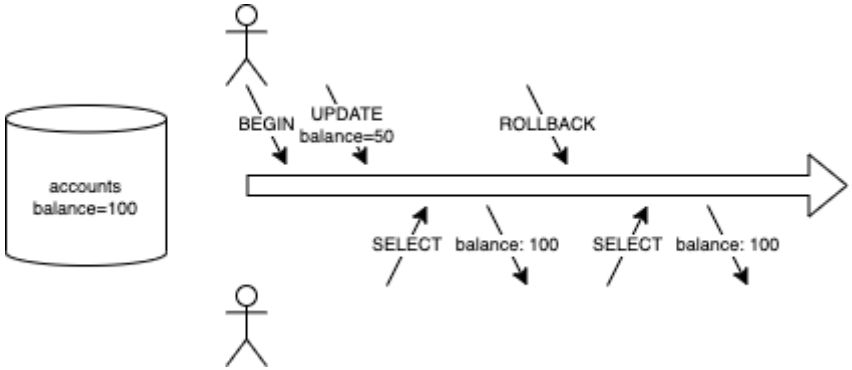
Atomicity

- Atomicity means that a transaction is always viewed as a single unit.
- It either succeeds completely.
- Or fails completely
- This also means that nobody ever sees the database in a partial state.

Atomicity



Atomicity



Consistency

- Consistency means that a database is always in a valid state.
- Valid can mean constraints, referential integrity, etc.
- When a transaction commits, the state must be valid.

Isolation

- Isolation means that when transactions run concurrently, they behave as if they’re run sequentially.
- Complicated subject, a lot of research in this area, but it’s a tool for concurrency control while maintaining data correctness.

Isolation levels

- 4 standard levels
    - Serializable: (As if) every transaction occurs sequentially.
    - Repeatable Read: A transaction only sees the data written before the transaction began, and successive SELECT queries see the same data.
    - Read Committed: A transaction only sees the data written before the transaction began.
    - Read Uncommitted: A transaction can see uncommitted updates from other transactions.

Isolation level phenomena

- Dirty read: A transaction reads data written by a concurrent uncommitted transaction.
- Nonrepeatable read: A transaction re-reads data it has previously read and finds that data has been committed by another transaction.
- Phantom read: Within a transaction the data returned from a second SELECT query is different from the first.
- Serialization anomaly: When 2 transactions produce a result which would be impossible if they were run sequentially.

Isolation levels vs phenomena

image::isolation_mysql.png

And now in Postgres

image::isolation_postgres.png

Performance and isolation levels

- Choosing an isolation has a huge effect on performance when there are many concurrent transactions.
- Serializable is typically way slower than the rest
- So do you choose speed or correctness?
- Every database is different here.

Durability

- When a transaction has been committed, the changes remain even after a system crash.
- Basically this means that everything is written to a log.

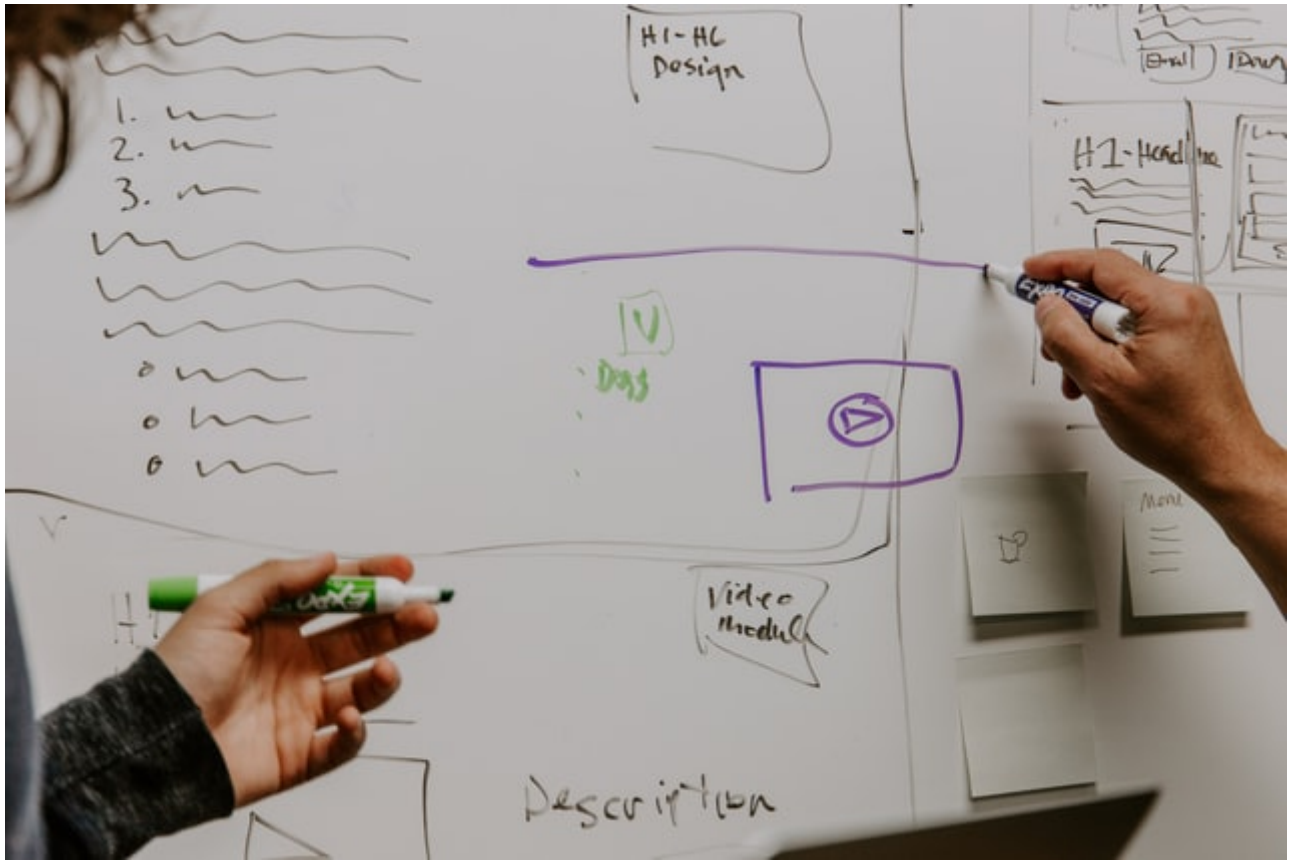
Other issues: Deadlock

- Transaction 1 waits for Transaction 2…
- while Transaction 2 waits for Transaction 1.
- Postgres has a "deadlock detection" feature, but this is not perfect.
- No magic fix for this. Requires deep understanding of the software.

Final words

- Transactions are very complex, and subject of much ongoing research.
- For now, all you need to know for now is the basics.
    - And that there is a lot of magic stuff going on under the hood.

Normalization



What is it?

- A data design technique used to:
    - reduce data redundancy,
    - ensure data is stored logically,
    - eliminate write anomalies.

The theory behind normalization

- Various "normal forms", named:
    - first normal form
    - second normal form
    - etc

The various normal forms

Let’s follow a tutorial:

https://www.guru99.com/database-normalization.html#3

Exercise Normalization

Create the sample database in:

java-traineeship-exercises/normalization.sql

Try to remodel towards BCNF.

Normalization advantages

- If executed properly, better data structure.
- Less duplication means data is less likely to be inconsistent.
- Reusing duplicate data might make the database smaller.
- A more flexible data design.

Normalization disadvantages

- Any "useful" query will need much more joins, reducing performance.
- More tables, means more keys, means more data to process.
- Often the 3rd normal form is practical.

Denormalization

- Duplicating data to reduce the amount of joins.
- Denormalized is not "not normalized".
- But explicit choices to inline data to improve performance of queries.
- Requires care when updating duplicated data!

Mongodb

Introduction

- Until now, we’ve only worked with relational databases.
- But there are a lot of other types.
- Some specialized, some general purpose.

NoSQL

- "Not Only SQL" or "No SQL".
- Often databases that sacrifice strictness for performance or scalability.
- Apache Cassandra works fine with 1000+ nodes in a cluster.
- Schemaless vs. schemas.

MongoDB

- MongoDB is a document database
- Meaning it stores documents in tables (called collections in MongoDB).

Big differences with relational databases

- Documents have a JSON-like structure: BSON which has some extras.
- No foreign keys
- Joins not very standard
- Transactions only introduced recently

So how can you build an application then?

- As opposed to a relational table, a document can contain a lot of structured data.
- Transactions are standard at the document level.
- You can still create foreign key relations, just not enforced by the database.
- Documents promote denormalization, which makes your app quicker.

But why?

- BSON maps well with objects.
- No enforced schema helps "schema" evolution, speeding up development.
    - Every data model has a "schema".

- It just works…
    - But you need to get rid of your relational mindset!

An example datamodel in MongoDB

TODO: Implement the student-lecture-course-program model in mongodb.

Example solution

- Note: Database designs in a document database can vary a lot!
- Design is done with normalization, but based on access paths.
- Keys and uniqueness are still important though!

Example solution 1/9

```
db.createCollection('students'); +
db.createCollection('courses'); +
db.createCollection('programs');
```

Example solution 2/9

```
db.programs.insert({
    "name" : "IT",
    "code" : "it",
    "courses" : []
})
```

Example solution 3/9

```
db.courses.insert({
    "name" : "Course 1",
    "requires" : [],
    "lectures" : []
})
```

Example solution 4/9

```
db.courses.update(
    {_id: ObjectId('course id')},
    {
        $push: {
            programs: "marketing"
        }
    }
)
```

Example solution 5/9

```
db.programs.update(
    {_id: ObjectId('program id')},
    {
        $push: {
            courses: {
                _id: ObjectId('course id'),
                name: 'Course 1'
            }
```

```
        }
    }
)
```

Example solution 6/9

```
db.students.insert({
    "name" : "Jane Doe",
    "birthdate" : ISODate("2022-05-01T07:19:02.493Z"),
    "attendsLectures" : [],
    "enrolledCourses" : []
})
```

Example solution 7/9

```
db.students.update(
{_id: ObjectId("student id")},
{
    $push: {
        "enrolledCourses" : {
            _id: ObjectId("course id"),
            name: "Course 1"
        }
    }
})
```

Example solution 8/9

```
db.students.update(
```

```
{_id: ObjectId("student id")},
{
    $push: {
        "attendsLectures" : {
            _id: ObjectId("lecture id"),
            name: "Lecture 1"
        }
    }
})
```

Example solution 9/9

- With this design it’s cheap to get the names of lectures, courses and programs.
- But at the cost of data duplication.
- Whether this is smart depends on the app!

Querying data

```
// Get all students that attended a lecture
db.students.find({"attendsLectures._id": ObjectId("lecture id")});
```

Applying projections

```
// Only return the name property, comparable to "SELECT name FROM students"
db.students.find({}, {name: 1});
```

Limit, offset and order by

```
// Sort by name, then skip the first 20 and only return max 10 results
db.students.find().skip(20).limit(10).sort({name: 1});
```

Indexes and performance 1/3

```
db.students.find({"attendsLectures._id": ObjectId()}).explain("executionStats")
// Collscan, so no index used
```

Indexes and performance 2/3

```
db.students.createIndex({"attendsLectures._id": 1}) // This also works in arrays
```

Indexes and performance 3/3

```
db.students.find({"attendsLectures._id": ObjectId()}).explain("executionStats")
// IXScan, so an index is used
```

Complex queries

- find() is limited.
- For complex queries, there is the Aggregation Framework

- Supports joins, group by, etc

Aggregation example

```
// Get the number of students per lecture
db.students.aggregate([
    {$unwind: "$attendsLectures"},
    {$group: {
        _id: "$attendsLectures._id",
        count: { $sum: 1 }
    }}
])
```

Aggregation example 2

```
// Get all students per lecture, not possible in SQL
db.students.aggregate([
    {$unwind: "$attendsLectures"},
    {$group: {
        _id: "$attendsLectures._id",
        studentIds: { $addToSet: "$_id" }
    }}
])
```

MongoDB summary

- MongoDB is a document database.
- Which supports richer models than relational databases.

- Transactions since version 4.
- But no foreign key constraints.
- Performs better and scales further than typical relational models, if applied correctly.
    - "Applying correctly" requires skill and care.