Veebirakenduse projekt (ITI0302)

Loeng 9

Siim Rebane

The problem

- The service calls `findAll(pageable)` or `findById(id)`; the database does not run Java
- The database receives **SQL** that Hibernate wrote: one or more statements per request
- Two things decide whether an endpoint is fast or slow:
  - **how many** statements one request sends → the entity mapping and the repository method
  - **how much work** each statement is → the table size and the indexes
- Neither can be seen in the Java code
  - the number of statements is read from the application's **log**
  - the work of one statement is read from the database, with `EXPLAIN`

The sample

- The staff application: departments and their employees
- `Employee` has a `@ManyToOne` `department`, **lazy**: loading an employee does not load its department
- `Department` has a `@OneToMany` list of employees
- `Project` has many employees, `Note` belongs to one employee
- Endpoints used in this lecture:
  - `GET /api/employees`: the list
  - `GET /api/employees/{id}`: one employee
  - `GET /api/employees/paged`: one page
  - `GET /api/employees/search`: one page, filtered and sorted
- The employee DTO carries the **department's name**, so the mapper reads `employee.getDepartment().getName()`
- Spring Boot 4.1.1, Hibernate 7.4, PostgreSQL 18 in Docker; 8 employees in 6 departments

`show-sql`

```
spring.jpa.show-sql=true
```

- Every statement Hibernate sends is printed to standard output, with `Hibernate:` in front

```
GET /api/employees
Hibernate: select e1_0.id,d1_0.id,d1_0.name,e1_0.email,e1_0.name,e1_0.phone,e1_0.start_date
           from employee e1_0 join department d1_0 on d1_0.id=e1_0.department_id
```

- This request made one statement
- `e1_0`, `d1_0`: Hibernate's aliases for the `employee` and `department` tables
- A parameter is printed as `?`
  - the statement is a **prepared statement**: sent with placeholders, the values sent separately
  - `show-sql` shows the placeholders only, not the values

Logging the statements with their values

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.orm.jdbc.bind=TRACE
spring.jpa.properties.hibernate.format_sql=true
```

- `org.hibernate.SQL=DEBUG` : the statements through the logging system, with timestamp, thread and level
- `org.hibernate.orm.jdbc.bind=TRACE` : one line per `?` , in order, with its type and value
- `format_sql=true` : one clause per line (compacted below)

```
GET /api/employees/search?departmentId=2&startedAfter=2021-01-01&sort=startDate,desc&size=2
05:32:41.700 DEBUG [nio-8089-exec-1] org.hibernate.SQL          :
    select e1_0.id, d1_0.id, d1_0.name, e1_0.email, e1_0.name, e1_0.phone, e1_0.start_date
    from employee e1_0 left join department d1_0 on d1_0.id=e1_0.department_id
    where e1_0.department_id=? and e1_0.start_date>?
    order by e1_0.start_date desc offset ? rows fetch first ? rows only
05:32:41.700 TRACE [nio-8089-exec-1] org.hibernate.orm.jdbc.bind : binding parameter (1:BIGINT) <- [2]
05:32:41.700 TRACE [nio-8089-exec-1] org.hibernate.orm.jdbc.bind : binding parameter (2:DATE) <- [2021-01-01]
05:32:41.701 TRACE [nio-8089-exec-1] org.hibernate.orm.jdbc.bind : binding parameter (3:INTEGER) <- [0]
05:32:41.701 TRACE [nio-8089-exec-1] org.hibernate.orm.jdbc.bind : binding parameter (4:INTEGER) <- [2]
05:32:41.732 DEBUG [nio-8089-exec-1] org.hibernate.SQL          :
    select count(e1_0.id) from employee e1_0 where e1_0.department_id=? and e1_0.start_date>?
```

- The thread name `nio-8089-exec-1` ties every line to one request
- The `left join department` comes from `@EntityGraph` on the repository method, not from a written join
- `offset ? rows fetch first ? rows only` : the page; skip `offset` rows, return `size` rows
- The `?` are numbered left to right; the bind lines come in that order, with the type and the value
- The second statement is Spring Data's count for `totalPages` , with the same `where`

Where to turn it on

- On the laptop, while looking at one endpoint
- Not in the committed `application.properties`; two ways that leave it alone:
  - command-line arguments after the jar: `--property=value`
  - IntelliJ: run configuration → Modify options → Program arguments, the same `--…` text

```
java -jar target/staff-0.0.1-SNAPSHOT.jar --logging.level.org.hibernate.SQL=DEBUG \
    --logging.level.org.hibernate.orm.jdbc.bind=TRACE
```

- Not on the server: every statement of every request, with its values, would go to the log
- The logger names are case-sensitive: `org.hibernate.SQL`, with capitals
  - the `LOGGING_LEVEL_…` environment-variable form lower-cases them and does not match

Counting the statements of an endpoint

- Send one request, count the `Hibernate:` lines
- The sample's endpoints, with the small data (8 employees, 6 departments):

Request	Statements	Why
`GET /api/employees`	1	`join fetch e.department`
`GET /api/employees/3`	2	`findById`, then the lazy `department` when the mapper reads its name
`GET /api/employees/paged?size=3`	2	the page, and the count
`GET /api/employees/search?departmentId=2`	1	the page; the count is skipped when the first page is not full
`GET /api/employees/search?name=an`	1	the same

- A count that grows with the data (7 statements for 6 employees) is the problem to look for
- A count that stays at 1 or 2 for any amount of data is fine
- PostgreSQL can also count statements: the `pg_stat_statements` extension keeps every statement with its count and time
  - needs enabling in the server's configuration; the application log is enough for this course

The N+1 problem

- The list endpoint loads all employees with one statement
- `department` is lazy, so no department is loaded with them
- The mapper then reads `getDepartment().getName()` for each employee; each read is one more statement
- `GET /api/employees`, six employees in six departments:

```
Hibernate: select e1_0.id,e1_0.department_id,e1_0.email,e1_0.name,e1_0.phone,e1_0.start_date from employee e1_0
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
```

- Seven statements: one for the list of N rows, N more for one relation of each row
- 500 employees → 501 round trips to the database
- The same with a `@OneToMany`: a department's employees, an order's lines, a post's comments
- Each of the N statements is fast: a lookup by primary key, a tenth of a millisecond
- The cost is the N round trips; on the laptop it is invisible, on the server, where the database is on another machine, it is not

The fix with `join fetch`

```java
@Query("select e from Employee e join fetch e.department")
List<Employee> findAllWithDepartment();
```

```
Hibernate: select e1_0.id,d1_0.id,d1_0.name,e1_0.email,e1_0.name,e1_0.phone,e1_0.start_date
           from employee e1_0 join department d1_0 on d1_0.id=e1_0.department_id
```

- One statement
- `join fetch` is JPQL: it names the relation to load together with the entity
- The same with `@EntityGraph(attributePaths = "department")` on a repository method; the SQL is a `left join`
- The same for a collection: `select d from Department d join fetch d.employees`
- Or a projection query that returns exactly the columns the DTO needs

A slow statement

- The log shows how many statements a request made, nothing about the work each one was
- The same statement, on 8 rows and on a million rows, looks identical in the log:

```
select … from employee e1_0 where e1_0.department_id=? and e1_0.start_date>? order by … fetch first ? rows only
```

- On 8 rows the database answers in a fraction of a millisecond
- On a million rows it reads the whole table, unless it has a way to go straight to the matching rows
- That way is an **index**
- Whether a statement has one is answered by the database, with `EXPLAIN`

Sequential scan and index scan

- A table is stored as **blocks** of 8 kB on disk, about 80 employee rows per block, rows in no particular order
- The block is what the database reads: never one row, always the whole block it is in
- To answer `where department_id = 500` the database has two options:
  - **sequential scan**: read every block of the table, test every row against the `where`
    - cost grows with the table: a million rows read, every time
  - **index scan**: look the value up in an index, read only the blocks that hold matching rows
    - cost grows with the number of matches, not with the table

An index

- A separate structure next to the table, a **B-tree**
  - the values of one or more columns, sorted, each with the location of its row
- The database keeps it up to date on every `insert`, `update` and `delete` of the table
- Every primary key has one, made with the constraint: `employee_pkey`
  - a `findById` is always an index scan
- Other columns have no index unless one is created
  - a foreign-key column is **not** indexed by PostgreSQL on its own
- The indexes of a table are listed under the table in IntelliJ's database tree
  - `employee` before this lecture: `employee_pkey` only, although `department_id` has a foreign key

EXPLAIN

- Before running a statement, PostgreSQL's **planner** chooses how: which scans, in what order, which joins
- `EXPLAIN` in front of the statement returns that plan instead of the rows

```
explain select e1_0.id, e1_0.name, e1_0.start_date from employee e1_0
    where e1_0.department_id = 500 and e1_0.start_date > date '2021-01-01'
    order by e1_0.start_date desc limit 2;
                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Limit  (cost=0.42..8.34 rows=2 width=27)
   ->  Index Scan Backward using idx_employee_department_start_date on employee e1_0  (cost=0.42..1789.46 rows=452 width=27)
         Index Cond: ((department_id = 500) AND (start_date > '2021-01-01'::date))
```

- `EXPLAIN <statement>`: the plan the planner chose, with its estimates; the statement is **not run**
- `ANALYZE` option: runs the statement and adds what really happened to every node: `actual time`, the real `rows`, `loops`
  - it runs the statement for real: an `EXPLAIN (ANALYZE) delete …` deletes
  - not the same as the `analyze employee` command, which updates the planner's statistics
- `BUFFERS` option: adds how many 8 kB blocks each node touched: `shared hit` from memory, `read` from disk
- `EXPLAIN (ANALYZE, BUFFERS)` to measure; plain `EXPLAIN` to see what the planner would do, without waiting for a slow statement
- The statement is the one from the log, with the `?` filled in from the bind lines
- It is run like any statement, in an SQL console connected to the development database (IntelliJ's database tool, for example)

Test data with `generate_series`

- On 8 rows every plan is fast; a million rows, generated by the database itself, in the SQL console:

```
insert into department (name) select 'Department ' || g from generate_series(7, 1000) g;
insert into employee (name, email, phone, start_date, department_id)
select 'Employee ' || g, 'employee' || g || '@example.com', null,
       date '2015-01-01' + (g % 4000), 1 + (g % 1000)
from generate_series(1, 1000000) g;
analyze employee;
```

```
INSERT 0 994 · INSERT 0 1000000 · Time: 11380.436 ms (00:11.380)
```

- `generate_series(1, 1000000)`: a function that returns rows, the numbers 1 to 1 000 000, one per row; it stands in the `from` like a table, `g` is its column
- `insert into … select …`: every row the `select` produces is inserted; no loop, no application code
- `'Employee ' || g`: text joined with the number, a different value per row
- `g % 1000` and `g % 4000`: the remainder spreads the rows over 1 000 departments and 4 000 start dates, so the filtered columns have many different values
- `date '2015-01-01' + (g % 4000)`: a date plus that many days
- The departments first, because `department_id` has a foreign key
- `analyze employee`: updates the statistics the planner uses; after a bulk load they describe the old, small table until this runs
- In the development database only
  - not a Liquibase changeset: the changelog describes the schema, not test data
  - not in the pipeline: the build job's database is thrown away after the tests

The plan before the index

- The search endpoint's statement: department 500, started after 2021, newest first, two rows
- 1 000 008 employees, only the primary key indexed

```
explain (analyze, buffers) select e1_0.id, e1_0.name, e1_0.start_date from employee e1_0
    where e1_0.department_id = 500 and e1_0.start_date > date '2021-01-01'
    order by e1_0.start_date desc limit 2;
 Limit  (cost=19587.95..19588.19 rows=2 width=27) (actual time=24.031..29.914 rows=2.00 loops=1)
   Buffers: shared hit=12412
   ->  Gather Merge  (cost=19587.95..19640.48 rows=451 width=27) (actual time=24.029..29.912 rows=2.00 loops=1)
         Workers Planned: 2   Workers Launched: 2
         ->  Sort  (cost=18587.93..18588.40 rows=188 width=27) (actual time=21.347..21.348 rows=2.00 loops=3)
               Sort Key: start_date DESC
               Sort Method: top-N heapsort  Memory: 25kB
               ->  Parallel Seq Scan on employee e1_0  (cost=0.00..18586.05 rows=188 width=27)
                                                       (actual time=0.147..21.208 rows=166.67 loops=3)
                     Filter: ((start_date > '2021-01-01'::date) AND (department_id = 500))
                     Rows Removed by Filter: 333169
                     Buffers: shared hit=12336
 Planning Time: 0.444 ms
 Execution Time: 29.972 ms
```

- Look at the innermost node first: `Parallel Seq Scan on employee`, every block of the table was read, by three workers
- `Rows Removed by Filter: 333169` per worker: a million rows read, 500 kept
- `Execution Time: 29.972 ms` for two rows

Reading the plan

- A plan is a tree, read from the innermost node outward: each node hands its rows to the one above

Part	Means
`cost=0.00..18586.05`	the planner's estimate: start-up cost and total cost, in units where reading one block in order is 1; not milliseconds; what the planner compares plans by
`rows=188 width=27`	estimated rows out of this node; average bytes per row
`(actual time=0.147..21.208 rows=166.67 loops=3)`	measured by `ANALYZE`: milliseconds to the first and to the last row, real rows, how many times the node ran (three workers; the numbers are per worker)
`rows=188` against `rows=166.67`	estimate against actual; a large gap means stale statistics, the `analyze` command fixes it
`Parallel Seq Scan on employee`	the node that reads the table: every block, in parallel workers
`Filter: (…)`, `Rows Removed by Filter: 333169`	the `where`, applied to each row after it was read; the rows read for nothing, per worker
`Sort … top-N heapsort`	the `order by`, keeping only the top 2 because of the `limit`
`Buffers: shared hit=12336`	blocks read from memory; `read=` would be blocks that came from disk; 12 336 × 8 kB is the whole 96 MB table
`Execution Time: 29.972 ms`	the measured total; `Planning Time` is what choosing the plan took

- `Seq Scan` on a large table with most rows removed by the filter: this statement has no index it could use

The index as a Liquibase changeset

- An index is part of the schema, so it is made like every schema change: a new changeset, applied at start

```
<changeSet id="006-employee-index" author="siim">
    <createIndex tableName="employee" indexName="idx_employee_department_start_date">
        <column name="department_id"/>
        <column name="start_date"/>
    </createIndex>
</changeSet>
```

```
liquibase.ui        : Running Changeset: db/changelog/changes/006-employee-index.xml::006-employee-index::siim
liquibase.changelog : Index idx_employee_department_start_date created
liquibase.changelog : ChangeSet db/changelog/changes/006-employee-index.xml::006-employee-index::siim ran successfully in 302ms
```

- The columns are the ones of the `where`, in the order the statement uses them: the equality first, the range second
- The `order by` is on the second column too, so the rows come out of the index already in that order
- The table's indexes afterwards: `employee_pkey` and `idx_employee_department_start_date (department_id, start_date)`

The column order of an index

- A B-tree over two columns is sorted by the first, and within equal values of the first by the second
  - like a phone book: by surname, then by first name
- The declared order decides which statements the index can serve

Index	Serves	Does not serve
`(department_id, start_date)`	`department_id = ?` alone; `department_id = ? and start_date > ?`; the same with `order by start_date`	`start_date > ?` alone: the first column is not fixed
`(start_date, department_id)`	`start_date > ?` alone	`department_id = ?` alone

- THE RULE: the columns compared with `=` first; the column with a range (`>`, `<`, `between`) or the `order by` last
- One two-column index serves the search statement and the department-only one
- The `startedAfter`-only filter would need its own

The plan after the index

- The same statement, the same million rows, the index in place:

```
Limit  (cost=0.42..8.34 rows=2 width=27) (actual time=0.053..0.057 rows=2.00 loops=1)
  Buffers: shared hit=5 read=3
  ->  Index Scan Backward using idx_employee_department_start_date on employee e1_0
          (cost=0.42..1789.46 rows=452 width=27) (actual time=0.051..0.056 rows=2.00 loops=1)
        Index Cond: ((department_id = 500) AND (start_date > '2021-01-01'::date))
        Index Searches: 1
        Buffers: shared hit=5 read=3
Planning Time: 0.683 ms
Execution Time: 0.080 ms
```

- `Index Scan Backward`: the index is read from its end, because the sort is descending, and stopped after two rows
- The `where` became an `Index Cond`: the rows are found through the index; the `where` is not tested on every row
- Eight blocks instead of 12 336
- The sort node is gone: the index delivers the rows in the wanted order
- No parallel workers: too little work to split

Before and after

	Before	After
plan	Parallel Seq Scan + Sort	Index Scan Backward
blocks read	12 412	8
rows read for nothing	999 507	0
execution time of the statement	29.97 ms	0.08 ms
`GET /api/employees/search?departmentId=500&…`, measured with `curl`	60 ms	10 ms
size on disk	table 96 MB	index 7 MB

- The endpoint's 10 ms that remain: the HTTP round trip, the count statement, Hibernate, JSON
- The Java code did not change; one changeset was added

Which columns get an index, and what it costs

Column	Index
the primary key	has one, from the constraint
a foreign key used in joins or filters: `department_id`, `employee_id`	yes; PostgreSQL does not make it for you
columns in the `where` of statements that run often	yes, in the order of the statement: equality columns first, then the range
the column of a frequent `order by`	yes, as the last column of such an index, so that the sort disappears
a column that is only ever selected, never filtered or sorted	no

- An index is not free: every `insert`, every `delete` and every `update` of an indexed column writes it too; it takes space, 7 MB here next to a 96 MB table
- The planner may still choose a sequential scan: when the `where` keeps most of the table, reading the blocks in order is cheaper
- So: `EXPLAIN (ANALYZE)` on the statements that run most, with realistic row counts; an index for every `Seq Scan` whose filter removes most rows; `EXPLAIN` again, because an index the planner does not use only costs

When an index cannot help

- The name search, `lower(name) like '%an%'`, on the million rows:

```
Limit  (cost=0.00..2.73 rows=20 width=23) (actual time=0.015..249.315 rows=2.00 loops=1)
  ->  Seq Scan on employee e1_0  (cost=0.00..27336.12 rows=200002 width=23) (actual time=0.014..249.313 rows=2.00 loops=1)
        Filter: (lower(name) ~~ '%an%'::text)
        Rows Removed by Filter: 1000006
Execution Time: 249.336 ms
```

- A B-tree is sorted by the value from its first character
- `'%an%'` can be anywhere in the value: the index cannot be searched for it
  - an index on `name` would not be used
  - an index on `name` does not serve `lower(name)` either; the index would have to be on `lower(name)`
- What helps a contains-search:
  - a prefix search, `'an%'`, with an index on `lower(name)`
  - the `pg_trgm` extension: indexes every three-letter piece of the text, serves `like '%an%'`
- For the course project the plain `like` on a few thousand rows is fine

Pagination, sorting and filtering

- Milestone 2 requires an endpoint with **pagination, sorting and search/filtering using Specifications**
- The sample's version:

```
GET /api/employees/search?name=an&departmentId=2&startedAfter=2021-01-01&page=0&size=2&sort=startDate,desc
```

Parameter	Does	Required
`name`	employees whose name contains the text, any case	no
`departmentId`	employees of one department	no
`startedAfter`	employees who started after the date	no
`page` , `size`	which slice, how big	no, defaults 0 and 20
`sort`	the field and the direction	no, default: the database's order

- Every filter is optional, and they combine: none, any one, any two, all three
- One endpoint, eight possible `where` clauses

Specifications

- A `where` **built at run time** from the filters that are present, in Java
- A repository method name cannot do it: its `where` is fixed, every value must be given; eight combinations would be eight methods
- A `@Query` cannot do it cleanly: one fixed string with `(:name is null or …)` for every filter; a new filter means editing the string
- Spring Data JPA's **Specification**: one object per condition, combined in Java, typed; built on the JPA Criteria API, Hibernate turns it into SQL with `?` parameters
- Part of `spring-boot-starter-data-jpa`; no new dependency

The controller

```
@GetMapping("/search")
public PageDto<EmployeeDto> search(@RequestParam(required = false) String name,
                                   @RequestParam(required = false) Long departmentId,
                                   @RequestParam(required = false) @DateTimeFormat(iso = ISO.DATE) LocalDate startedAfter,
                                   Pageable pageable) {
    return employeeService.search(new EmployeeFilter(name, departmentId, startedAfter), pageable);
}
```

```
public record EmployeeFilter(String name, Long departmentId, LocalDate startedAfter) {}
```

- `required = false`: an absent parameter is `null`
- `@DateTimeFormat(iso = ISO.DATE)`: reads `2021-01-01` into a `LocalDate`
- The filters go to the service as one record, `null` for the absent ones
- `Pageable`: Spring fills it from `page`, `size` and `sort` of the request

The service

```
public PageDto<EmployeeDto> search(EmployeeFilter filter, Pageable pageable) {
    Page<Employee> page = employeeRepository.findAll(EmployeeSpecifications.matching(filter), pageable);
    return new PageDto<>(employeeMapper.toDtoList(page.getContent()),
            page.getNumber(), page.getTotalPages(), page.getTotalElements());
}
```

```
public record PageDto<T>(List<T> items, int page, int totalPages, long totalElements) {}
```

- One repository call: the Specification made from the filter, and the `Pageable`
- `Page<Employee>` comes back: the slice of entities, and the total count
  - two statements: the slice with the `where`, and `select count(…)` with the same `where`
- The entities are mapped to DTOs as always
- The answer is a `PageDto` of our own: the items, the page number, the total pages, the total elements
  - `Page` itself is not returned; its JSON shape is Spring's and changes between versions

The Specifications class

```
public final class EmployeeSpecifications {

    public static Specification<Employee> matching(EmployeeFilter f) {
        return Specification.allOf(nameContains(f.name()), inDepartment(f.departmentId()),
                                   startedAfter(f.startedAfter()));
    }

    public static Specification<Employee> nameContains(String part) {
        return (root, query, cb) -> part == null ? null
                : cb.like(cb.lower(root.get("name")), "%" + part.toLowerCase() + "%");
    }

    public static Specification<Employee> inDepartment(Long departmentId) {
        return (root, query, cb) -> departmentId == null ? null
                : cb.equal(root.get("department").get("id"), departmentId);
    }

    public static Specification<Employee> startedAfter(LocalDate date) {
        return (root, query, cb) -> date == null ? null
                : cb.greaterThan(root.get("startDate"), date);
    }
}
```

- One static method per filter; each returns `null` when its value is absent: no condition
- `Specification.allOf(…)` : joins the conditions with `and` , skips the `null` ones; `anyOf` joins with `or` ; `a.and(b)` , `a.or(b)` , `Specification.not(a)` combine two

One Specification, line by line

```
public static Specification<Employee> inDepartment(Long departmentId) {
    return (root, query, cb) -> {
        if (departmentId == null) return null;                    // null = no condition
        return cb.equal(root.get("department").get("id"), departmentId);
    };
}
```

- `Specification<Employee>` has one method, `toPredicate(root, query, cb)`; a lambda with the three arguments is one
- `root`: the entity in the `from`; `root.get("name")` is the column of the field `name`
  - `root.get("department").get("id")` walks the relation: Hibernate makes it `e1_0.department_id = ?`, without a join
- `query`: the whole query being built; rarely needed
- `cb`, the `CriteriaBuilder`: makes the conditions: `cb.equal(…)`, `cb.like(…)`, `cb.greaterThan(…)`
- The result is a `Predicate`, one condition of the `where`; **`null` means no condition**
- The value goes in as a `?` parameter, never into the SQL text

`CriteriaBuilder` operations

Call	SQL
`cb.equal(root.get("x"), v)`	`x = ?`
`cb.notEqual(root.get("x"), v)`	`x <> ?`
`cb.like(root.get("x"), "%v%")`	`x like ?`
`cb.lower(root.get("x"))`	`lower(x)`, an expression to use in another call
`cb.greaterThan`, `lessThan`, `greaterThanOrEqualTo`, `lessThanOrEqualTo`	`x > ?`, `x < ?`, `x >= ?`, `x <= ?`
`cb.between(root.get("x"), a, b)`	`x between ? and ?`
`cb.isNull(root.get("x"))`, `isNotNull`	`x is null`, `x is not null`
`root.get("x").in(list)`	`x in (?, ?, ?)`
`cb.and(p1, p2)`, `cb.or(p1, p2)`, `cb.not(p)`	`(… and …)`, `(… or …)`, `not (…)`

- The names are the **entity's** field names: `startDate`, not `start_date`
- A wrong name fails at the first call, not at compile time

A range with optional ends

- A filter from–to, where either end may be missing:

```
public static Specification<Employee> startedBetween(LocalDate from, LocalDate to) {
    return (root, query, cb) -> {
        if (from == null && to == null) return null;
        if (from == null) return cb.lessThanOrEqualTo(root.get("startDate"), to);
        if (to == null) return cb.greaterThanOrEqualTo(root.get("startDate"), from);
        return cb.between(root.get("startDate"), from, to);
    };
}
```

- Four cases, one Specification, one condition or `null`

The repository

```
public interface EmployeeRepository extends JpaRepository<Employee, Long>,
                                            JpaSpecificationExecutor<Employee> {

    @EntityGraph(attributePaths = "department")
    Page<Employee> findAll(Specification<Employee> spec, Pageable pageable);
}
```

- `JpaSpecificationExecutor<Employee>` adds the methods that take a Specification:
  - `findAll(spec)`, `findAll(spec, pageable)`, `findAll(spec, sort)`
  - `findOne(spec)`, `count(spec)`, `exists(spec)`, `delete(spec)`
- `findAll(spec, pageable)` is declared again so that `@EntityGraph` can be put on it: the departments are loaded with the page, in the slice statement

The statements the search endpoint made

```
GET /api/employees/search?departmentId=2                                    1 statement
Hibernate: select e1_0.id,d1_0.id,d1_0.name,e1_0.email,e1_0.name,e1_0.phone,e1_0.start_date
           from employee e1_0 left join department d1_0 on d1_0.id=e1_0.department_id
           where e1_0.department_id=? offset ? rows fetch first ? rows only
```

```
GET /api/employees/search?name=an                                           1 statement
Hibernate: select … from employee e1_0 left join department d1_0 on d1_0.id=e1_0.department_id
           where lower(e1_0.name) like ? escape '' offset ? rows fetch first ? rows only
```

```
GET /api/employees/search?departmentId=2&startedAfter=2021-01-01&sort=startDate,desc&size=2   2 statements
Hibernate: select … from employee e1_0 left join department d1_0 on d1_0.id=e1_0.department_id
           where e1_0.department_id=? and e1_0.start_date>? order by e1_0.start_date desc
           offset ? rows fetch first ? rows only
Hibernate: select count(e1_0.id) from employee e1_0 where e1_0.department_id=? and e1_0.start_date>?
```


```
{"items":[{"id":8,"name":"Laura White","email":"laura.white@example.com","departmentName":"Engineering"},
          {"id":7,"name":"Chris Evans","email":"chris.evans@example.com","departmentName":"Engineering"}],
 "page":0,"totalPages":1,"totalElements":2}
```


- The same endpoint, a different `where` for each set of parameters
- The first two have no count statement: the matches fit in the default page of 20, so the total is known from the slice

Sorting

Parameter	Form	Becomes
`sort=startDate,desc`	entity field, direction	`order by e1_0.start_date desc`
`sort=name`	field only	`order by e1_0.name` ascending
`sort=departmentId&sort=name`	twice	`order by … department_id, … name`
no `sort`		no `order by`: the database's order, not stable across pages

- The field in `sort` is the **entity's** field name
- An unknown field name fails when the request comes in, not when the code compiles
  - an endpoint that exposes sorting checks the field, or answers 400
- A paged list needs a stable order, so that page 2 does not repeat a row of page 1
  - a default sort, `id` at least, when the request gives none
- `size` is capped by `spring.data.web.pageable.max-page-size`, default 2 000; a request for `size=100000` gets 2 000