01 - access paths, joins

Zadání dotazů

Spočtěte ceny (v počtu I/O operací čtení) následujících dotazů:

- **SELECT * FROM EMP WHERE EMPNO = 745**
  1. bez použití indexu
  2. s použitím indexu
- **SELECT * FROM EMP WHERE EMPNO > 150**
  1. bez indexu
  2. s indexem
  3. s cluster indexem
- **SELECT * FROM EMP WHERE EMPNO < 151**
  1. bez indexu
  2. s indexem
  3. s cluster indexem
- **SELECT * FROM DEPT WHERE DEPTNO = 40**
  1. bez indexu
  2. s indexem
- **SELECT * FROM DEPT D, EMP E WHERE D.DEPTNO = E.DEPTNO**
  i. pro M = 3
  ii. pro M = 4
  iii. pro M >= 5
- Předpokládejme indexově organizovanou tabulku EMP. Kolik stojí dotaz **SELECT * FROM EMP WHERE EMPNO=1546**.
- Předpokládejme uložení EMP a DEPT ve společném indexovaném clusteru X s klíčem DEPTNO.
  - Počet různých hodnot klíče je 100. Index má stejnou kvalitu jako byl index nad DEPT(DEPTNO)
  - V každém fragmentu clusteru je jeden řádek z DEPT a (při splnění předpokladu rovnoměrného rozložení distinct hodnot) 300 řádků z EMP, zhruba bX ~ bEMP ~ 60
  - I(X,DEPTNO) = I(DEPT, DEPTNO) = 1
    i. **SELECT * FROM EMP E, DEPT D WHERE E.DEPTNO = D.DEPTNO AND D.DEPTNO=30**
    ii. **SELECT * FROM EMP WHERE EMPNO = 1745**
    iii. **SELECT * FROM EMP WHERE DEPTNO = 30**
- EMP není uložena v clusteru, ale nad EMP.DEPTNO existuje index. Index na EMP.DEPTNO:
  - Bi ~ 150
  - Fi (počet větvení ve vnitřním bloku indexu), fI = 100.
  - I(EMP,DEPTNO) (hloubka stromu) ~ log(card(EMP.DEPTNO))/log(Fi) ~ log(300) / log(100) = 2.48/2 ~ 2
  - Jaká je cena **SELECT * FROM EMP WHERE DEPTNO = 30**?

Řešení

Úkol 1: dopočet statistik

- **EMP**:
  - Bemp (blokovací faktor) cca 60
  - Pemp (počet stránek pro uložení relace) = 30 000 / 60 ~ 500
  - Index nad EMPNO:
  - I(EMP,EMPNO) (hloubka stromu) ~ log(Nr)/log(Fi) ~ log(30 000) / log(100) = 4.47/2 ~ 5/2 ~ 3
- **DEPT**:
  - Bdept ~ 40
  - Pdept = 100 / 40 ~ 3
  - Index nad DEPTNO:
    - I(DEPT, DEPTNO) ~ log(100) / log(100) = 2/2 = 1

Zadání dotazů

Spočtěte ceny (v počtu I/O operací čtení) následujících dotazů:

- **SELECT * FROM EMP WHERE EMPNO = 745**
  1. bez použití indexu
     - cena = Pemp = 500 (uvažován nejhorší případ)
  2. s použitím indexu
     - cena = I(EMP, EMPNO) + 1 = 3 + 1 = 4
- **SELECT * FROM EMP WHERE EMPNO > 150**
  1. bez indexu
     - cena = Pemp = 500
  2. s indexem
     - cena = cesta dolu indexem + cesta doprava listovými uzly+jednotlivé návštěvy datových bloků = I(EMP, EMPNO) + [(MAX(EMPNO)-hranice)/Bi]+ (MAX(EMPNO)-hranice) = 3+ (30000-150)/150+ (30000-150)=3+29850/150+29850=3+199+29850 = 30052 > 500
  3. s cluster indexem

    - cena = … datové řádky sousedních klíčů jsou vedle sebe v datových blocích = 3+29850/150+[29850/Bemp]=3+199+29850/60=3+195+498~ 696 > 500
- **SELECT * FROM EMP WHERE EMPNO < 151**
  1. bez indexu
     - cena = Pemp= 500
  2. s indexem
     - cena = cesta dolu indexem + cesta doleva listovými uzly+jednotlivé návštěvy datových bloků = I(EMP, EMPNO) + [(hranice - MIN(EMPNO))/Bi]+ (hranice - MIN(EMPNO) = 3+(150)/150 + (150)=3+1+150= 153 < 500
  3. s cluster indexem
     - cena = … datové řádky sousedních klíčů jsou vedle sebe v datových blocích, cena = 3+150/150+[150/Bemp]=3+1+150/60 ~ 7 < 500
- **SELECT * FROM DEPT WHERE DEPTNO = 40**
  1. bez indexu
     - cena = pR =3
  2. s indexem
     - cena = I(DEPT, DEPTNO) + 1 = 1 + 1 = 2
- **SELECT * FROM DEPT D, EMP E WHERE D.DEPTNO = E.DEPTNO**
  - IPr - PsI je velký, použijeme hnízděné cykly, vnější bude relace DEPT
    1. pro M = 3
       - jedna stránka bude použita jako výstupní buffer, jedna jako buffer pro čtení DEPT a jedna jako buffer pro čtení EMP. Cena = Pdept + Pdept*Pemp = 3 + 3*500 = 1503. Při využití „dopředu a dozadu“ ve vnitřním cyklu cena nižší o dvě čtení v úvrati = 1501
    2. pro M = 4
       - jedna stránka bude použita jako výstupní buffer, jedna jako buffer pro čtení EMP a dvě jako buffer pro čtení DEPT. Cena = Pdept + (Pdept/2)*Pemp = 3+2*500 = 1003
    3. pro M >= 5
       - jedna stránka bude použita jako výstupní buffer, jedna jako buffer pro čtení EMP a tři jako buffer pro čtení DEPT. Všechny stránky DEPT se tedy mohou načíst do paměti a tudíž se tabulka EMP bude číst pouze jednou. Cena = Pdept + (Pdept/3)*Pemp = 3 + 1*500 = 503
- Předpokládejme indexově organizovanou tabulku EMP. Kolik stojí dotaz **SELECT * FROM EMP WHERE EMPNO=1546**.
  - cena = I(EMP, EMPTNO) = 3. POZNÁMKA: U těchto struktur musíme uvažovat délku řádku v listu (příliš dlouhé řádky lze rozdělit do zvláštního extentu).
- Předpokládejme uložení EMP a DEPT ve společném indexovaném clusteru X s klíčem DEPTNO.
  - Počet různých hodnot klíče je 100. Index má stejnou kvalitu jako byl index nad DEPT(DEPTNO)
  - V každém fragmentu clusteru je jeden řádek z DEPT a (při splnění předpokladu rovnoměrného rozložení distinct hodnot) 300 řádků z EMP, zhruba bX ~ bEMP ~ 60
  - I(X,DEPTNO) = I(DEPT, DEPTNO) = 1
    1. **SELECT * FROM EMP E, DEPT D WHERE E.DEPTNO = D.DEPTNO AND D.DEPTNO=30**
       - cena = I(X,DEPTNO) + Nemp(DEPTNO=30) / bX = 1 + (30 000 / 100) / 60 = 1 + 5 = 6. POZNAMKA: Je vidět, že spojení nás při uložení v clusteru nic nestojí …
    2. **SELECT * FROM EMP WHERE EMPNO = 1745**
       - cena = I(EMP, EMPNO) + 1 = 3 + 1 = 4. POZNÁMKA: Nad clusterem můžeme mít i jiné indexy, než je klíč clusteru a EMPNO je PK relace EMP.
    3. **SELECT * FROM EMP WHERE DEPTNO = 30**
       - cena = I(X, DEPTNO) + Nemp(DEPTNO=30) / Bx = 1 + (30 000 / 100) / 60 = 6. POZNAMKA: cena provedení dotazu je stejná, jako při spojení s DEPT
- EMP není uložena v clusteru, ale nad EMP.DEPTNO existuje index. Index na EMP.DEPTNO:
  - Bi ~ 150
  - Fi (počet větvení ve vnitřním bloku indexu), fI = 100.
  - I(EMP,DEPTNO) (hloubka stromu) ~ log(card(EMP.DEPTNO))/log(Fi) ~ log(300) / log(100) = 2.48/2 ~ 2
  - Jaká je cena **SELECT * FROM EMP WHERE DEPTNO = 30**?
    - Cena = I(EMP, DEPTNO)+ (Nemp(DEPTNO=30)/Bi -1) + Nemp(DEPTNO=30) = 2 + 1 + (30 000 / 100) ~303
    - POZNAMKA: Index nad sloupcem DEPTNO v relaci EMP je trochu „nešťastný“ neboť jeho selektivita je nízká (pro jednu hodnotu indexu DEPTNO dostaneme při rovnoměrném rozložení 300 (30 000 /100) indexových položek pro stejnou hodnotu klíče. Ty při blokovacím faktoru bI =150 zaberou tři zřetězené bloky v listu indexu. V nejnepříznivější případě každé ROWID ukazuje do jiného bloku segmentu tabulky. Cena v nejnepříznivějším případě je tedy dána počtem řádků se stejnou hodnotou klíče DEPTNO.

02 - execution plans

query Q1

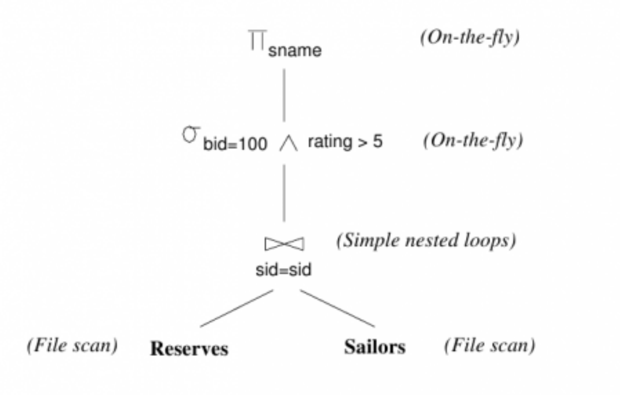
```sql
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
       AND R.bid = 100 AND S.rating > 5
```

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

In „our“ simplified notation:

```
(sailor * reservers) (bid = 100 and rating > 5) [sname]
```

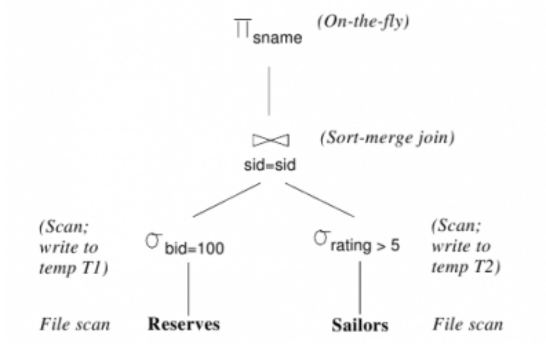
query Q1 plan P1



- Reserves 40 bytes long, page 100 Reserves tuples, and 1,000 pages.
- Sailors 50 bytes long, page 80 Sailors tuples, and 500 pages.
- M = 3 (minimal memory requirements !!)

Cost of the join is 1, 000 + 1, 000 $*$ 500 = 501, 000 page I/Os. Selections and projection on-the-fly - **total cost = 501, 000**

query Q1, plan P2 - pushing selections



query Q1, plan P2 - cost 1/3

- selection bid=100 (assume uniform distribution, 100 boats)
  - 1000 pages for scan + 10 pages for writing selection result (T1)
- selection rating > 5 (assume uniform distribution, rating ranges 1..10)
  - 500 pages for scan + 250 for writing selection result (T2)

Selection cost = 1 000 + 10 + 500 + 250 = **1760** I/O pages

query Q1, plan P2 - cost 2/3

> here, suppose M=5 (recalculation for M=3 to be consistent with previous example is obvious)

- merge join - only 5 buffer pages
  - T1 sorting - 10 pages - 2 passes
    - first pass 2*5 reading + 2*5 writing
    - second pass 2*5 reading + 10 writing
    - cost = 2*2*10 = 40
  - T2 sorting - 250 pages - 4 passes
    - cost = 2*4*250 = 2 000
  - reading T1 and T2 for merging
    - cost = 250 + 10 = 260

Merge join cost = 40 + 2 000 + 250 + 10 = **2 300** I/O pages

Remark: Using priority queue may reduce the cost of sorting T2 even more.

query Q1, plan P2 - cost 3/3

- projection is done on-the-fly (no additional I/O operations)

Q1, P2 total cost = selection cost + merging cost = 1760 + 2300 = **4 060 I/O pages**

query Q1, plan P3 - cost 1/1

Suppose to use nested loop join instead of merge join.

- nested loop join
    - T1 10 pages - outer relation
    - T2 250 pages - inner relation
    - M=5 in buffer: 3 for T1, 1 for T2, out buffer: 1
    - NL cost = 10 + (10*250)/3 = **830 (#I/O reads)**
        - pR+pRpS/(M-2) (#I/O reads)

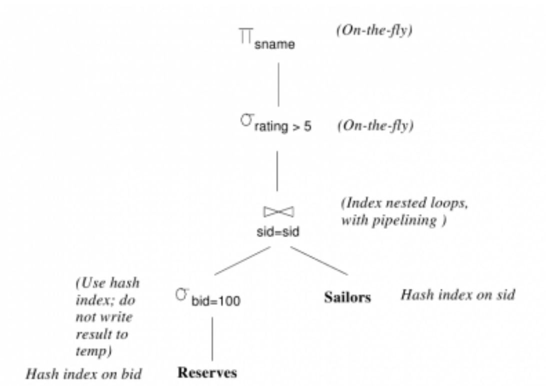
Q1 cost = 1 760 + 840 = **2 600 I/O pages**

query Q1, plan P4 - cost 1/1

Projection pushing

- reduces size of intermediate relations T1 an T2
- T1 reduces to sid - 3 pages only (instead of 10)
- T2 reduces to sid, sname
- NL cost < 250 I/O pages
- total cost of Q1 for plan P4 **< 2 000 I/O pages**

query Q1, plan P5



query Q1, plan P5 - cost 1/1

Suppose special structures.

- Reservs: 100 000 / 100 = 1 000 tupples with BID=100
    - clustered index 1000 tupples ~ 10 pages
- Sailors - hash index on SID - direct access for 1000 tuples ~ 1000 I/O pages

Rest (selection, projection) on-the-fly.

Total cost Q1,P5 ~ 1010 I/O pages.

query Q1, plan P6 - cost 1/1

further improvements:

- Sailors stored in cluster
- Reserves clustered indexed on bid
- materialize (projeted) seletion on Reserves
- sort it and join with Sailors by sid access
- seletion to rating on-the-fly

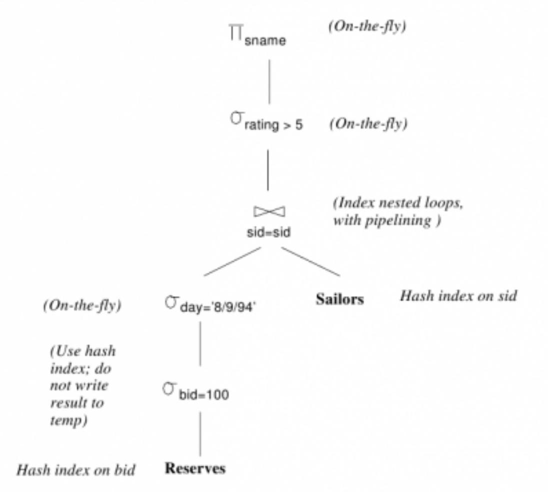
Materialization brings benefit here.

query Q2 (additional selection)

```sql
SELECT S.sname
FROM Reserves R, Sailors S
```

```sql
WHERE R.sid = S.sid
      AND R.bid = 100
      AND S.rating > 5
      AND R.day = '8/9/94';
```

query Q2 - plan P1



query Q2, plan P1 - cost 1/1

- bid,day,sid - clustered index on Reservs
    - bid, day selection, projection to sid → T1
    - T1 sorting
    - no need to access Reservs (only index scan)
    - cost C1 = I(reservs,(bid,day,sid)) + pages_to_store_wanted_keys + sorting
        - ~ 3 + 1 + 1 = 5 (~ 100 tuples)
- Sailors is clustered by sid
    - cost C2 = I(sailor,sid) + pages_to_store_wanted_sailor_range
        - ~ 2 + 3
- selection on rating on-the-fly

total cost ~ **10 I/O pages**

```
// --------------------------------------------------------------------
// #1
// Find actors born in 1966 with first name Jiri

db.actors.find(
    { year: 1966, "name.first": "Jiri" }
).pretty();

// --------------------------------------------------------------------
// #2
// Find movies directed by Jan Hrebejk
// Note that the order of fields for first and last names is arbitrary

db.movies.find(
    { "director.first": "Jan", "director.last": "Hrebejk" }
).pretty();

// --------------------------------------------------------------------
// #3
// Find actors with first name Jiri who played in Medvidek movie
// Return names of these actors only

db.actors.find(
    { "name.first": "Jiri", movies: "medvidek" },
    { name: 1, _id: 0 }
).pretty();

// --------------------------------------------------------------------
// #4
// Find movies shot between years 2000 and 2005 such that they have a
director specified
// • Return movie identifier only
// • Order the result by ratings in descending order and then by years
in ascending order

db.movies.find(
    {
        year: { $gte: 2000, $lte: 2005 },
        director: { $exists: 1 }
    },
    { _id: 1 }
).sort(
    { rating: -1, year: 1 }
).pretty();

// --------------------------------------------------------------------
// #5
// Find actors who stared in Samotari or Medvidek movies, return id

db.actors.find(
    { movies: { $in: [ "medvidek", "samotari" ] } },
    { _id: 1 }
).pretty();

db.actors.find(
    { $or: [ { movies: "medvidek" }, { movies: "samotari"} ] },
    { _id: 1 }
).pretty();
```

```
// --------------------------------------------------------------------
// #6
// Find actors who played in both Samotari and Medvidek
// • Return actor identifier only
// • Find two different approaches

db.actors.find(
    { movies: { $all: [ "medvidek", "samotari" ] } },
    { _id: 1 }
).pretty();

db.actors.find(
    { $and: [ { movies: "medvidek" }, { movies: "samotari" } ] },
    { _id: 1 }
).pretty();


// --------------------------------------------------------------------
// #7
// Find movies with Czech title equal to Vratne lahve
// • Return movie title only
// • Note that there are two means how movie titles are defined

db.movies.find(
    { $or: [
        { title: "Vratne lahve" },
        { "title.cs": "Vratne lahve" }
    ] },
    { title: 1, _id: 0 }
).pretty();

db.movies.find(
    { $or: [
        { title: { $eq: "Vratne lahve" } },
        { "title.cs": { $eq: "Vratne lahve" } }
    ] },
    { title: 1, _id: 0 }
).pretty();

// --------------------------------------------------------------------
// #8
// Find actors having their movies defined as an array
//  • Return actor identifier and the second and third movie only (if
any)

db.actors.find(
    {
        $or: [
            { movies: { $size: 0 } },
            { "movies.0": { $exists: 1 } }
        ]
    },
    { _id: 1, movies: { $slice: [ 1, 2 ] } }
).pretty();

// alternatively it is possible to use $type (or its alias "array")
// https://docs.mongodb.com/manual/reference/operator/query/type/
```

```
// ---------------------------------------------------------------------
// #9
// Find movies in which at least one actor with an identifier ending
with ova played
// • Return movie identifier and the first actor who satisfies such a
condition

db.movies.find(
    { actors: { $regex: /ova$/ } },
    { "actors.$": 1 }
).pretty();

// ---------------------------------------------------------------------
// #10
// Find movies that have a Czech Lion award from 2005
// • Return movie idenθfier and all awards

db.movies.find(
    {
        awards: { $elemMatch: { type: "Czech Lion", year: 2005 } }
    },
    { _id: 1, awards: 1 }
).pretty()

/*INCORRECT*/
db.movies.find(
    { "awards.type": "Czech Lion", "awards.year": 2005 },
    { _id: 1, awards: 1 }
).pretty()

// ---------------------------------------------------------------------
// #11
// Find movies that are comedies and dramas at the same time or have a
rating 80 or more
// • Return movie identifier only

db.movies.find(
    {
        $or: [
            { genres: { $all: [ "comedy", "drama" ] } },
            { rating: { $gte: 80 } }
        ]
    },
    { _id: 1 }
).pretty();

// ---------------------------------------------------------------------
---------
```

4. Cvičení - Neo4j

Příklad 1 - filmy, herci

Možné řešení

```
//------------------------------------------------------------------------------
// #1
// Find movies with identifier medvidek. Return movie nodes together with title properties.


MATCH (m:MOVIE {id: "medvidek"})
RETURN m, m.title;

MATCH (m:MOVIE)
  WHERE m.id = "medvidek"
RETURN m, m.title;

//------------------------------------------------------------------------------
// #2
// Find actors born in 1965 or later. Return actor names and years they were born.
// Sort the result using years (in descending order)and then names (in ascending order)


MATCH (a:ACTOR)
  WHERE a.year >= 1965
RETURN a.name, a.year
  ORDER BY a.year DESC, a.name ASC;

... ORDER BY a.year DESCENDING, a.name ASCENDING;

//------------------------------------------------------------------------------
// #3
// Find titles of movies in which Jiri Machacek played.


MATCH (:ACTOR {name: "Jiri Machacek"})<--(n:MOVIE)
RETURN n.title;

MATCH (:ACTOR {name: "Jiri Machacek"})--(n:MOVIE)
RETURN n.title;

MATCH (:ACTOR {name: "Jiri Machacek"})<-[:PLAY]-(n:MOVIE)
RETURN n.title;

MATCH (n:MOVIE)-[:PLAY]->(:ACTOR {name: "Jiri Machacek"})
RETURN n.title;

MATCH (n:MOVIE)-[:PLAY]->(a:ACTOR)
  WHERE a.name = "Jiri Machacek"
RETURN n.title;

//------------------------------------------------------------------------------
// #4
// Find movies where at least one actor played.


MATCH (m:MOVIE)-[:PLAY]->(:ACTOR)
RETURN DISTINCT m;

MATCH (m:MOVIE)
  WHERE EXISTS( (m)-[:PLAY]->(:ACTOR) )
RETURN m;

MATCH (m:MOVIE)
  WHERE (m)-[:PLAY]->(:ACTOR)
RETURN m;

MATCH (m:MOVIE)
   WHERE SIZE([p = (m)-[:PLAY]->(:ACTOR) | p] )  >= 1
RETURN m;

//------------------------------------------------------------------------------
// #5
// Find actors who played with Ivan Trojan.


MATCH
  (s:ACTOR {name: "Ivan Trojan"})
    <-[:PLAY]-(m:MOVIE)-[:PLAY]->
  (a:ACTOR)
RETURN DISTINCT a;

MATCH
  (s:ACTOR {name: "Ivan Trojan"})<-[:PLAY]-(m:MOVIE),
  (m)-[:PLAY]->(a:ACTOR)
RETURN DISTINCT a;

MATCH (s:ACTOR {name: "Ivan Trojan"})<-[:PLAY]-(m:MOVIE)
MATCH (m)-[:PLAY]->(a:ACTOR)
```

```
  WHERE a <> s
RETURN DISTINCT a;

MATCH (a:ACTOR)
  WHERE
    SIZE( [p =
      (a)<-[:PLAY]-(:MOVIE)-[:PLAY]->(:ACTOR {name: "Ivan Trojan"} )
     |p] ) >= 1
RETURN a;

//-------------------------------------------------------------------------------
// #6
// Find all friends of actor Ivan Trojan include friends of friends etc.  Return actor names.


MATCH (s:ACTOR {name: "Ivan Trojan"})-[:KNOW *]-(a:ACTOR)
  WHERE s <> a
RETURN DISTINCT a.name;

MATCH (s:ACTOR {name: "Ivan Trojan"})-[:KNOW *1..]-(a:ACTOR)
  WHERE s <> a
RETURN DISTINCT a.name;

MATCH (a:ACTOR)
  WHERE
    EXISTS( (a)-[:KNOW *]-(:ACTOR {name: "Ivan Trojan"}) )
    AND
    (a.name <> "Ivan Trojan")
RETURN a.name;

//-------------------------------------------------------------------------------
// #7
// Find pairs of movies and their actors. Include movies without actors as well.


MATCH (m:MOVIE)
OPTIONAL MATCH (m)-[:PLAY]->(a:ACTOR)
RETURN m.title, a.name;

//-------------------------------------------------------------------------------
// #8
// Find actors who played in movies having above average number of actors. Return actor names.

MATCH (m:MOVIE)
WITH m, SIZE([q = (m)-[:PLAY]->(:ACTOR) | q ]) AS actors
WITH AVG(actors) AS average
MATCH (m)
  WHERE SIZE( [p = (m)-[:PLAY]->(:ACTOR) | p]) > average
MATCH (m)-[:PLAY]->(a:ACTOR)
RETURN DISTINCT a.name;
//-----------------------------------------------------------------------------------
```

Příklad 2 - letiště

Databáze

Několik príkladů

```
//Find nodes with the code SF (2 alternatives)

//start clause is deprecated from version 4.0

start n=node(*) where n.code='sf' return n;

match (s{code:'sf'}) return s;

//Find all direct flights from SF (2 alternatives)

//start clause is deprecated from version 4.0

start s=node(*) match (s)-[:DIRECT]->(d)
   where s.code='sf' return s,d;

match (s{code:'sf'})-[:DIRECT]->(d) return s,d;

//Find all flights from SF of max length 5 (display paths)

//extract function is deprecated from version 3.5

match path=(s{code:'sf'})-[:DIRECT*1..5]->(d)
   return  extract(x in nodes(path) |x.code);

// syntax from version 3.5 uses list comprehension:

match path=(s{code:'sf'})-[:DIRECT*1..5]->(d)
   return  [x in nodes(path) |x.code];

// Flights starting in SF, ending in NY, max length 5, display paths and prices of paths

match
```

```
 path=(s{code:'sf'})-[:DIRECT*1..5]->(d{code:'ny'})
return
 extract(x in nodes(path) |x.code) as total_path,
 reduce(acc=0, x in relationships(path)|acc+x.price)
   as total_price;

//with list comprehension:

match
 path=(s{code:'sf'})-[:DIRECT*1..5]->(d{code:'ny'})
return
 [x in nodes(path) |x.code] as total_path,
 reduce(acc=0, x in relationships(path)|acc+x.price)
   as total_price;


//Flights starting in SF, ending in NY, max length 5, display paths and prices of paths, order output by 

// function extract is deprecated in neo4j 4
// does not work in neo4j 5

match
 path=(s{code:'sf'})-[:DIRECT*1..5]->(d{code:'ny'})
return
 extract(x in nodes(path) |x.code) as total_path,
 reduce(acc=0, x in relationships(path)|acc+x.price)
   as total_price
order by total_price
limit 3;

//with list comprehension
// new syntax neo4j 5

match
 path=(s{code:'sf'})-[:DIRECT*1..5]->(d{code:'ny'})
return
 [x in nodes(path) |x.code] as total_path,
 reduce(acc=0, x in relationships(path)|acc+x.price)
   as total_price
order by total_price
limit 3;
```

XPath

- Names of all airline companies (whole airline elements)

```
/airport/lines/line/airline
/child::airport/child::lines/child::line/child::airline
//line/airline
//airline
```

- Full names of all airports (just text content)

```
//airport/text()
/descendant-or-self::node()/airport/text()
/descendant-or-self::airport/text()

//line/*/airport/text()
//line/*[name() = "departure" or name() = "arrival"]/airport/text()
```

- Codes of all airports (their values)

```
data(//airport/@code)
data(//child::airport/attribute::code)
```

- The last ticket of the third flight (in the document order)

```
/airport/flights/flight[3]/tickets/ticket[last()]
/airport/flights/flight[position() = 3]/tickets/ticket[position() =
last()]
```

- Distinct codes of flight ticket classes (without duplicities)

```
distinct-values(/airport/flights/flight/tickets/ticket/@class)
distinct-values(//ticket/@class)
distinct-values(//@class)
distinct-values(data(//@class))
```

- Flight numbers operated by A6-EOQ aircraft on 2017-10-13

```
/airport/flights/
  flight[@date = "2017-10-13"][aircraft/@registration = "A6-EOQ"]/
  line

/airport/flights/
  flight[@date = "2017-10-13" and aircraft/@registration = "A6-EOQ"]/
  line
```

- Flights with at least one first class ticket (F) or business class ticket (C)

```
//flight[tickets/ticket/@class = "F" or tickets/ticket/@class = "C"]

//flight[.//@class = "F" or .//@class = "C"]

//flight[tickets/ticket[@class = "F" or @class = "C"]]

//flight[
  count(tickets/ticket[@class = "F"]) >= 1
  or
  count(tickets/ticket[@class = "C"]) >= 1
]

//flight[.//@class = ("F", "C")]

(: INCORRECT :)
//flight[//@class = "F" or //@class = "C"]
```

- Flights without any first class ticket (F) as well as any business class ticket (C). Include only flghts with at least one ticket.

```
(: INCORRECT :)
//flight[.//ticket][.//@class != "F" and .//@class != "C"]

(: CORRECT :)
//flight[.//ticket][not(.//@class = "F" or .//@class = "C")]

(: CORRECT :)
//flight[count(.//ticket) >= 1][
  count(.//ticket[@class = "F"]) = 0
  and
  count(.//ticket[@class = "C"]) = 0
]

(: INCORRECT :)
//flight[count(.//ticket) >= 1][
  count(.//ticket/@class = "F") = 0
  and
  count(.//ticket/@class = "C") = 0
]
```

- Numbers of flights that depart on 2017-10-18 or any date later and that have no aircraft assigned yet.

```
count(//flight[@date >= "2017-10-18" and not(aircraft)]/line)
count(//flight[attribute::date >= "2017-10-18" and
not(child::aircraft)]/line)
count(//flight[./@date >= "2017-10-18" and not(./aircraft)]/line)
count(//flight[self::node()/@date >= "2017-10-18" and
not(self::node()/aircraft)]/line)
count(//flight[(@date >= "2017-10-18") and not(aircraft)]/line)
count)//flight[@date >= "2017-10-18"][not(aircraft)]/line)
count(//flight[not(aircraft)][@date >= "2017-10-18"]/line)
count(//flight[@date >= "2017-10-18"][count(aircraft) = 0]/line)

count(//flight/line[../@date >= "2017-10-18" and not(../aircraft)])
count(//flight/line[parent::node()/@date >= "2017-10-18" and
not(parent::node()/aircraft)])
count(//flight/line[parent::flight/@date >= "2017-10-18" and
not(parent::flight/aircraft)])
```

- Lines with duration above the overall average.

```
//line[duration > avg(//line/duration)]
//line[duration > avg(//duration)]
//line[duration/text() > avg(//line/duration/text())]
```

- Overall number of flights heading to any airport in Germany (DEU) on 2017-10-18

```
count(
  //flight
    [@date = "2017-10-18"]
    [line = //line[arrival/airport/@country = "DEU"]/@number]
)
```

- Passenger name on the very last ticket in the entire file.

```
(: INCORRECT :)

//ticket[last()]/text()
//ticket[position() = last()]/text()
/descendant-or-self::node()/ticket[position() = last()]/text()

(: CORRECT :)

/descendant::ticket[last()]/text()
/descendant-or-self::ticket[last()]/text()

//ticket[not(following::ticket) and not(descendant::ticket)]/text()
//ticket[not(following::ticket)]/text()

(//ticket)[last()]/text()
```

XQuery

- Flights heading to any airport in Germany (DEU) on 2017-10-18

```
//flight
  [@date = "2017-10-18"]
  [line = //line[arrival/airport/@country = "DEU"]/@number]

let $a := //flight[@date = "2017-10-18"][line =
//line[arrival/airport/@country = "DEU"]/@number]
return $a

let $a := //line[arrival/airport/@country = "DEU"]/@number
return //flight[@date = "2017-10-18"][line = $a]

let $a := //line[arrival/airport/@country = "DEU"]/@number
for $f in //flight[@date = "2017-10-18"][line = $a]
return $f

let $a := //line[arrival/airport/@country = "DEU"]/@number
for $f in //flight
where ($f/@date = "2017-10-18") and ($f/line = $a)
return $f
```

- Sequence of lines longer than 60 minutes

```
(: 2A :)

for $n in /airport/lines/line
where $n/duration > 60
return
  <line origin="{ $n/departure/airport/@code }" destination="{
$n/arrival/airport/@code }">
    <code>{ data($n/@number) }</code>
    <departure>{ data($n/departure/time) }</departure>
    <arrival>{ data($n/arrival/time) }</arrival>
  </line>

  ...<line origin="{ data($n/departure/airport/@code) }" ...>

  ...<departure>{ $n/departure/time/text() }</departure>
  ...<arrival>{ $n/arrival/time/text() }</arrival>

for $n in /airport/lines/line[duration > 60]
...

(: INCORRECT :)

... <code>{ $n/@number }</code>
... <departure>{ $n/departure/time }</departure>

(: -----------------------------------------------------------------------
------ :)
(: 2B :)

...
return
  element line {
    attribute origin { $n/departure/airport/@code },
    attribute destination { $n/arrival/airport/@code },
    element code { data($n/@number) },
    element departure { data($n/departure/time) },
    element arrival { data($n/arrival/time) }
  }
```

- Names of airline companies such that all their flights are associated with aircrafts.

```
(: 3 :)

for $a in distinct-values(//airline)
let $n := //line[airline = $a]/@number
where
  every $f in //flight[line = $n] satisfies $f/aircraft
return $a

for $a in distinct-values(//airline)
let $n := //line[airline = $a]/@number
where
  every $l in $n satisfies
    every $f in //flight[line = $l] satisfies $f/aircraft
return $a
```

```
for $a in distinct-values(//airline)
let $n := //line[airline = $a]/@number
where
  count(//flight[line = $n]) = count(//flight[line = $n][aircraft])
return $a
```

- Generate an XHTML table with data about ﬂights from PRG.

```
(: 4 :)

<table>
  <tr><th>Date</th><th>Time</th><th>Number</th><th>Aircraft</th></tr>
  {
    let $n := //line[departure/airport/@code = "PRG"]/@number
    for $f in //flight[line = $n]
    let $t := //line[@number = $f/line]/departure/time/text()
    order by $f/@date descending, $t ascending
    return
      <tr>
        <td>{ data($f/@date) }</td>
        <td>{ $t }</td>
        <td>{ $f/line/text() }</td>
        <td>
          {
            if ($f/aircraft)
            then data($f/aircraft/@registration)
            else <i>Unknown</i>
          }
        </td>
      </tr>
  }
</table>
```

- Names of passengers of EK140 flights with at least average number of sold tickets over all EK140 flights.

```
(: 5 :)

let $c :=
  for $f in //flight[line = "EK140"]
  return count($f/tickets/ticket)
let $a := avg($c)
for $f in //flight[line = "EK140"]
where count($f/tickets/ticket) >= $a
return
  <passengers date="{ $f/@date }" tickets="{ count($f/tickets/ticket)
}">
    { string-join($f/tickets/ticket/text(), ", ") }
  </passengers>

...
return
  <passengers tickets="...">
    { $f/@date }
    ...
  </passengers>
```

```
let $p :=
  for $f in //flight[line = "EK140"]
  return
    <passengers date="{ $f/@date }" tickets="{ count($f/tickets/ticket)
}">
      { string-join($f/tickets/ticket/text(), ", ") }
    </passengers>
let $a := avg($p/@tickets)
return $p[@tickets >= $a]

(: -----------------------------------------------------------------
------ :)
```