Optimization criteria – analysis and metrics

- execution plan has assigned a **cost**
  - CPU – small in comparison to data access
  - data access – either in memory buffer (cache) or directly in secondary storage (disk, disk array)
- consider **only data access** and **no cache subsystems**
  - in practice, the influence of database buffer cache is important
  - logical vs physical readings
- data access – reading/writing done by **pages (blocks) of a unified size**

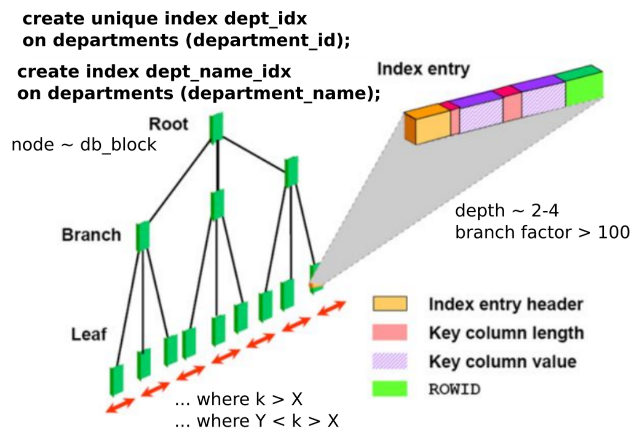
cost of an execution plan

means **# of I/O blocks to be processed**

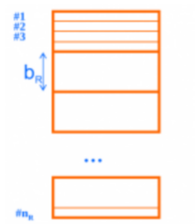
statistics

- about relation and indexes must be maintained
- used to guess/compute cost of individual operations and the whole execution plan

B-tree - reminder

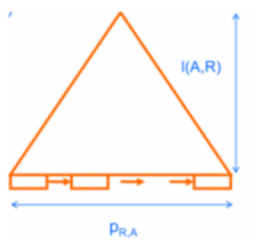


Basic table and index statistics



(heap) table statistics

nR	# of tuples (rows) of relation R
V(A,R)	# of R[A] (# of different values A in relation R)
pR	# of pages to store R
	(how many tuples(rows) in average fit into a block)
bR	block factor



b-tree statistics

Let's have index on relation R with key A.

f(A,R)	average # of followers (children) of branch node
	(~50-150 in real DBs)
I(A,R)	# levels of index tree (index tree depth)
	(~2-3 in real DBs) ~ log(V(A,R))/log(f(A,R))
p(A,R)	# of leafs blocks for index tree

Selection cost with base table approach

```
select * from R where A = 'x';
```

no index (full-table scan of heap table)

- cost = pR/2 if A is unique (unique constraint force index creation)
- **cost = pR** if A is non-unique

unique index on R(A)

- cost = I(A,R) + 1 in case of heap table + index
  - primary key constraint creates unique index
- cost = I(A,R) in case of IOT

non-unique index on R(A)

- cost = I(A,R) + n(R(A='x'))
  - in practice highly depends on clustering factor of index

Selection cost with index only approach

```sql
select B from R where A = 'x';
```

composed index on R(A,B)

- cost = I(R,(A,B)) + n(R(A=‘x’))/bI(A,B)-1
  - -1 because walking down the index tree we already reached the first leaf block of index

composed index on R(A,B), but A is unique

- cost = I(R,(A,B))
  - looks strange, but may be a design decision

Selection cost of non-equal selection

index query only

```sql
select A from R where A < 'x';
```

- cost = I(A,R) + pI(A,R)/2
  - evaluate A = ‘x’, take all leaf blocks on left side (ordering list of leafs)
  - average cost (we suppose value ‘x’ is in middle)

base table query

```sql
select * from R where A < 'x';
```

- cost = pR without using index R(A)
- cost = I(A) + pI(R,A)/2 +nR/2 with using index R(A)
  - averege cost

Selection – example 1/2

specification

- **sailor**(sid, sname, rating, age), labeled S
- **reservation**(bid,sid,day, remark), labeled R

nR	10 000	# of tuples/records
bR	50	block factor for R
V(sid,R)	500	# of dif. val. of sid in R
V(bid,R)	50	# of dif. val. of bid in R
f(I((bid,sid,day),R))	20	branch factor of index tree
bI((bid,sid,day),R)	20	block factor for index

calculation of pR

pR = nR / bR

pR	200	# of blocks to store table reservation

Selection – example 2/2

```sql
select sid, date from reservation where bid = 77;
```

full table scan (FTS) approach

cost = pR = 200 (we have to read the whole table)

using index on (bid,sid,day)

Suppose **even destribution of bid**; usually unreal in practice.

- cost = I((bid,sid, day),R) + n(R(bid=77)/bI - 1
  - V(R,bid) = 50, f(I(bid,sid,day),R) = 20
  - I((bid,sid,day),R) ~ log(50)/log(20) ~ 2
  - nR(bid=77) = nR/V(bid,R) = 10 000 / 50 = 200 rows
  - nR(bid=77)/bR = 200/20 = 10
- cost = 2 + 10 - 1 = 11 (10x faster then FTS)
- whole evaluation was done on index only (no base table access)

Multi-run sort examples 1/3

Sorting in RDBMS

- for merge-sort
- also for DISTINCT, ORDER BY, HAVING, set operations
- # of in memory / 2 run / multi run sorting is important system statistic

Example one: pR=300, M=50; do not consider priority queue

- 2-runs sorting
  1. run: partial sorting: 6 sorted pieces of length 50 (M=50)
  2. run: merging pieces together in M=7 (one for each piece, one for output)
- cost: 4pR I/O (2pR for reading, 2pR for writing)
  1. run: read, in-memory sort, write i.e. pR reading, pR writing
  2. run: read, merge, store i.e. pR reading, pR writing

Multi-run sort example 2/3

- pR= 5 000, M=50, without priority queue
- 3-runs sorting
  1. run 100 pieces of length 50
  2. run merging pieces: M: 49 for reading + 1 for output
     - the first sub-result has length 49x50
     - the second sub-result merges next 49 pieces from the first run
     - the third sub-result merges the last 2 pieces (100 - 49 - 49 = 2)
     - we got three sorted pieces of lengths 49x50, 49x50, 2x50
  3. run merging three pieces from 2. run together got one sorted piece of length 5 000
- cost: 6pR I/O = #runs∗pR reading + #runs∗pR writing = 2∗#runs∗pR I/O

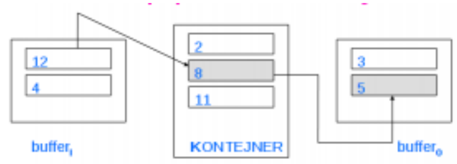
Multi-row sort example 3/3

- pR= 5 000, M=50 with priority queue
- evaluation
  1. run - 50 pieces of length 100 (guess for priority queue: piece approx. 2*M)
  2. run merging pieces: M: 49 reading + 1 for output
     - the first sub-result of length 49x100
     - one piece of length 100 remainds (50 - 49 = 1)
     - we got 2 sorted pieces of length 49x100 and 100 a the end of 2. run
  3. run merging three pieces from 2. run together (M=3 is enough)
- cost: 6pR -200 I/O = 3pR -100 reading + 3pR - 100 writing
  1. run pR reading pR writing
  2. run merging pR - 100 pages: pR - 100 reading, pR - 100 writing
  3. run merging pieces pR reading, pR writing

Priority queue – improvment of sorting

- read imput and sort it into runs
- qurantees that **avarege length of run is 2M**

1. split memory into three areas
2. read input buffer and fill container
3. from container select lowest value grater than top of output buffer
4. fill container from input buffer
5. repeat from step 3; if output buffer is full, write the run



Join evaluation – overview

Join methods (each RDBMS engine implements more of them)

- nested loops join
- merge join
- hash join
- join using special structures (index lookup, common cluster, . . .)

Assumptions

- `R(a,...),S(k, a, ...)` ; 1:N relationship
- join is mostly done by equality, usually natural join `R*S`
- cost calculation suppose
  - **# of memory blocks to process join** labelled **M**

Remind yourself relational algebra operations:

$$*, \times, R[\Theta]S, [\alpha], (\phi), \cap, \cup, -$$

Nested loops join – idea

Idea

```
foreach page r in R do
  foreach page s in S do
    find matched tuples in [r,s], put them to output
```

- consider only **# or I/O blocks to be processed**
- i.e. comparision of individual tuples of r and s is done in memory
  - cost=0

Discussion according to M

- M = 3 (minimal amount of memory)
- M >= pR+2 (optimal variant; suppose pR <= pS)
- M < pR+2

Nested loops join - cost according to M

suppose pR <= pS

M=3 (minimal memory amount)

- pR + pR∗Ps; #I/O read
- (nR∗nS/V(a,S))/bRS; #I/O write

M >= pR + 2

- pR + pS ; #I/O read

M in (3, pR + 2)

- pR + pR∗pS/(M-2); #I/O read

Merge join

```
select * from R join S on (R.a=S.a);
```

1. sort R according to a
2. sort S according to a
3. merge: read sorted relations, if R.a=S.a then construct result

- sorting means read-sort-write
- sorting usually does not fit into memory
  - more “runs” - i.e. read-sort-write several times
  - length of run either priority queue or M (see discussion bellow)

Merge join – cost (only #I/O read is discussed)

M=3

- cost ~ 2pR∗log(pR) + 2pS∗log(pS) + pR + pS

M >= sqrt(pS)

- create a sorted “runs” of length approx. 2*M
  - use priority queue structure to construct runs
  - at most sqrt(pS) runs for both pR and pS
- merging stage
  - allocate one page for each run
  - read runs in parallel and merge them together
- cost ~ 3(pR+pS)

M < sqrt(pS)

- depends on # of runs
- discussed in following examples

Join using indexes and special structures

R is ordered by a, S.a is primary key

- cost: pR + I(A,S) +p(S,A) + V(A,R)
  - read R (pR) and index on S(A) (I(A,S)+(p(S,A))
  - lookup to S by rowid in case of equality (V(A,R))

R is ordered by a, S is hashed by A (i.e. in hash cluster by a)

- cost: pR + V(A,R)

Join + selection, R.b is primary key, S.a is secondary key

```
select * from R join S on (R.a=S.a) where R.b='x';
```

- cost: I(a,S) + I(b,R) + 2

Find Operation: Examples

Select all movies from our collection

```
db.movies.find()
```

```
db.movies.find( { } )
```

Select a particular movie based on its document identifier

```
db.movies.find( { _id: ObjectId("2") } )
```

Select movies filmed in *2000* with a rating greater than *1*

```
db.movies.find( { year: 2000, rating: { $gt: 1 } } )
```

Select movies filmed between *2005* and *2015*

```
db.movies.find( { year: { $gte: 2005, $lte: 2015 } } )
```


Value Equality

Example (revisited)

Select movies having a specific director

```
db.movies.find(
  { director: { firstname: "Jan", lastname: "Svěrák" } }
)
```

```
db.movies.find(
  { "director.firstname": "Jan", "director.lastname": "Svěrák" }
)
```


Query Operators

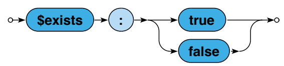
Comparison operators

- `$eq`, `$ne`
  - Tests the actual field value for **equality** / **inequality**
    - The same behavior as in case of value equality conditions
- `$lt`, `$lte`, `$gte`, `$gt`
  - Tests whether the actual field value is **less than** / **less than or equal** / **greater than or equal** / **greater than** the provided value
- `$in`
  - Tests whether the actual field value is equal to **at least one** of the provided values
- `$nin`
  - Negation of `$in`

Query Operators

Element operators

- `$exists` – tests whether a given field **exists** / **not exists**



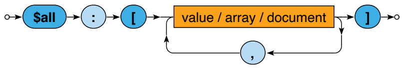
Evaluation operators

- `$regex` – tests whether a given field value matches a specified **regular expression** (PCRE)
- `$text` – performs **text search** (text index must exists)

Query Operators

Array operators

- `$all` – tests whether a given array **contains all the specified items** (in any order)



Example (revisited)

Select movies having specific actors

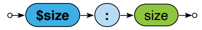
```
db.movies.find(
  { actors: [ ObjectId("5"), ObjectId("7") ] }
)
```

```
db.movies.find(
  { actors: { $all: [ ObjectId("5"), ObjectId("7") ] } }
)
```

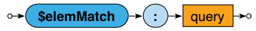

Query Operators

Array operators (cont'd)

- `$size` – tests the size of a given array against a fixed number (and not, e.g., a range, unfortunately)



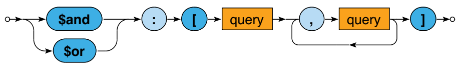
- `$elemMatch` – tests whether a given array **contains at least one item** that satisfies all the involved query operations



Query Operators

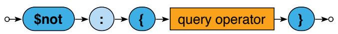
Logical operators

- `$and`, `$or`



  - Logical connectives for **conjunction** / **disjunction**
  - At least 2 involved query expressions must be provided
  - **Only allowed at the top level** of a query

- `$not`



  - Logical **negation** of exactly one involved query operator
  - I.e. **cannot be used at the top level** of a query

Querying Arrays

Condition based on **value equality** is satisfied when...

- the given field as a whole is identical to the provided value, or
- at least one item of the array is identical to the provided value

```
db.movies.find( { actors: ObjectId("5") } )
```

```
{ actors: ObjectId("5") }
```

```
{ actors: [ ObjectId("5"), ObjectId("7") ] }
```

Querying Arrays

Condition based on **query operators** is satisfied when...

- the given field as a whole satisfies all the involved operators, or
- each of the involved operators is satisfied by at least one item of the given array
  - note, however, that this item **may not be the same** for all the individual operators

```
db.movies.find( { ratings: { $gte: 2, $lte: 3 } } )
```

```
{ ratings: 3 }
```

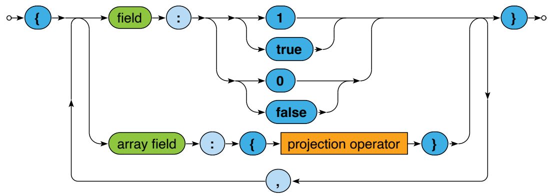
```
{ ratings: [ 3, 7, 5 ] }
```

```
{ ratings: [ 1, 4 ] }
```

Use `$elemMatch` when just a single array item should be found for all the operators

Projection

Projection allows us to determine the fields returned in the result

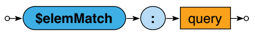


- `true` or `1` for fields to be **included**
- `false` or `0` for fields to be **excluded**
- Positive and negative enumerations cannot be combined!
  - The only exception is `_id` which is **included by default**
- **Projection operators** – allow to select particular array items

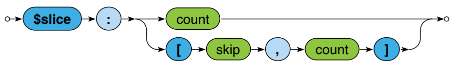
Projection Operators

Array operators

- `$elemMatch` – selects the first matching item of an array
  This item must satisfy all the operators included in `query`
  When there is no such item, the field is not returned at all



- `$slice` – selects the first `count` items of an array (when `count` is positive) / the last `count` items (when negative)
  Certain number of items can also be skipped



Projection: Examples

Find a particular movie, select its identifier, title and actors

```
db.movies.find(
  { _id: ObjectId("2") },
  { title: true, actors: true }
)
```

```
{
  _id: ObjectId("2"),
  title: "Samotáři",
  actors: [ ObjectId("6"),
            ObjectId("4"),
            ObjectId("5") ]
}
```

Find movies from *2000*, select their titles and the last two actors

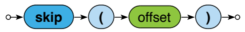
```
db.movies.find(
  { year: 2000 },
  {
    title: 1, _id: 0,
    actors: { $slice: -2 }
  }
)
```

```
{
  title: "Samotáři",
  actors: [ ObjectId("4"),
            ObjectId("5") ]
}
```

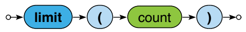
Modifiers

Modifiers change the order and number of returned documents

- `sort` – orders the documents in the result
- `skip` – skips a certain number of documents from the beginning



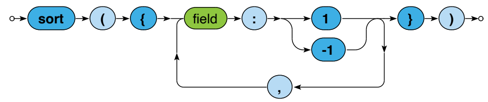
- `limit` – returns at most a certain number of documents



All the modifiers are optional, can be chained in **<u>any order</u>** (without any implications), but **must all be specified before any documents are retrieved** via a given cursor

Modifiers

Sort modifier orders the documents in the result



- 1 for **ascending**, −1 for **descending** order
- The order of documents is undefined unless explicitly sorted
- Sorting of larger datasets should be supported by indices
- **Sorting happens before the projection phase**
  - I.e. not included fields can be used for sorting purposes as well

Sample Query

Names of actors who played in *Medvídek* movie

```
MATCH (m:MOVIE)-[:PLAY]->(a:ACTOR)
  WHERE m.title = "Medvídek"
RETURN a.name, a.year
  ORDER BY a.year
```

m	a
(medvidek)	(trojan)
(medvidek)	(machacek)

→

a.name	a.year
Ivan Trojan	1964
Jiří Macháček	1966

Clauses and Subclauses

Read clauses

- `MATCH` – describes graph pattern to be searched for
  - `WHERE` – adds additional filtering constraints

Write clauses

- `CREATE`, `DELETE`, `SET`, `REMOVE`
  - Creates / deletes nodes / relationships / labels / properties

General clauses

- `RETURN` – defines what the query result should contain
  - `ORDER BY`, `SKIP`, and `LIMIT` subclauses
- `WITH` – constructs auxiliary intermediate query result
  - `ORDER BY`, `SKIP`, `LIMIT`, and also `WHERE`

Path Patterns

Node pattern

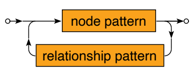
- Describes one data **node** and conditions it must satisfy
  - E.g.: `()`, `(:ACTOR { name: "Ivan Trojan" })`, …

Relationship pattern

- Describes one data **relationship** and conditions it must satisfy
  - E.g.: `()--()`, `(:MOVIE)-[:PLAY]->(:ACTOR)`, …

Path pattern

- Describes one data **path** to be found
  - Via a sequence of interleaved node and relationship patterns



Node Patterns

Node pattern (cont'd)

- **Labels condition**
  - Set of zero or more labels can be provided
  - Data node to be matched then…
    - Must have at least **all the specified labels**
    - I.e., there may also be other, but these are compulsory
  - E.g.: `(m:MOVIE)`
- **Property map condition**
  - Data node to be matched…
    - Must have at least **all the specified properties**
    - I.e., they are present and have **identical values**
    - Note that mutual order of such properties is unimportant
  - E.g.: `(m:MOVIE { title: "Medvídek", year: 2007 })`

Relationship Patterns

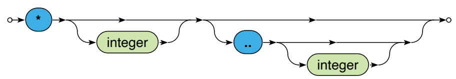
Relationship pattern (cont'd)

- **Type condition**
  - Set of zero or more types can be provided
  - Data relationship to be matched then…
    - Must have **one of the enumerated types**
  - E.g.: `()-[r:PLAY]->()`
- **Property map condition**
  - Data relationship to be matched…
    - Must have at least **all the specified properties**
  - E.g.: `()-[r:PLAY { role: "Jakub" }]->()`

Relationship Patterns

Relationship pattern (cont'd)

- **Variable length mode**
  - When activated, **paths of arbitrary lengths** can be found
    - Otherwise (i.e., by default), one relationship pattern will be matched by exactly one data relationship
  - **Length condition** ranges: `*`, `*4`, `*2..6`, `*..6`, `*2..`



  - Each data relationship on the path must…
    - **Satisfy all the involved conditions** (direction, type, properties)
  - E.g.: `()-[r:FRIEND *..2]-()`
    - If **variable** is introduced, it then references the whole path

Graph Patterns

Relationship uniqueness requirement

- **One data node** may match **multiple node patterns** at once
  - E.g.: `(a)-[:FRIEND]-()-[:FRIEND]-(b)`
    - It may happen that both `a` and `b` will actually be the same node
    - However, only when **distinct data relationships** were used…
- I.e., **one data relationship** cannot be matched repeatedly

Node pattern alignment

- Intentional alignment of nodes (not relationships) is possible
  - Simply by using the same **shared variables**
- E.g.: `(a)-[:FRIEND]-()-[:FRIEND]-(a)`

General graph pattern

- Graphs can be decomposed into individual path patterns
  - Uniqueness requirement / shared variables work the same way

Match Clause: Example

Names of actors who played with *Ivan Trojan* in any movie

- Notice that *Ivan Trojan* himself is not included in the result
  - Because of the **uniqueness requirement**

```
MATCH (i:ACTOR)<-[:PLAY]-(m:MOVIE)-[:PLAY]->(a:ACTOR)
  WHERE (i.name = "Ivan Trojan")
RETURN a.name
```

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

i	m	a
(trojan)	(samotari)	(machacek)
(trojan)	(samotari)	(schneiderova)
(trojan)	(medvidek)	(machacek)

→

a.name
"Jiří Macháček"
"Jitka Schneiderová"
"Jiří Macháček"

Match Clause: Example

Names of actors who played with *Ivan Trojan* in any movie (cont'd)

- **Uniqueness requirement** is <u>not</u> applied **across clauses**
  - And so internal identities must be used to exclude *Ivan Trojan*

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

i	m
(trojan)	(samotari)
(trojan)	(medvidek)

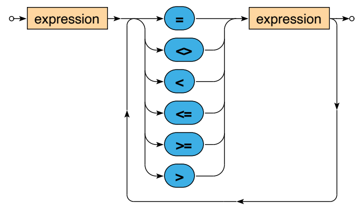
⋈

m	i
(vratnelahve)	(machacek)
(vratnelahve)	(sverak)
(samotari)	(trojan)
(samotari)	(machacek)
(samotari)	(schneiderova)
(medvidek)	(trojan)
(medvidek)	(machacek)

Search Conditions

Comparison conditions

- Traditional comparison operators are available
  - **Chained comparisons** can be created, too

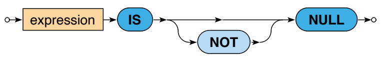


- E.g.: `2015 <= m.year < 2020`
  - Equivalent to `2015 <= m.year AND m.year < 2020`

Search Conditions

NULL testing conditions

- **Three-valued logic** is assumed
  - Traditional `true` and `false` values
    - But also `null` representing the third **unknown value**
- Indirect testing is thus necessary



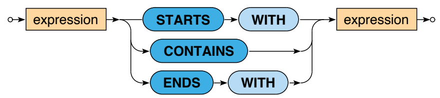
IN predicate conditions

- Allow for both **fixed enumerations** as well as **arbitrary lists**
- E.g.: `m.language IN [ "cs", "sk" ]`
- E.g.: `"comedy" IN m.genres`

Search Conditions

String matching conditions

- `STARTS WITH` / `CONTAINS` / `ENDS WITH` operators



- E.g.: `m.title ENDS WITH "Bobule"`
  - Matches `Bobule`, `2Bobule`, …

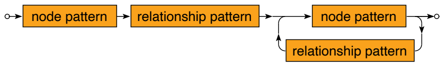
Regular expression conditions

- Special operator `=~` is used for this purpose
- E.g.: `m.title =~ ".*Bobule"`

Search Conditions

Path pattern predicate conditions

- **Path pattern** can directly be used as a condition
  - At least one relationship pattern is required, though

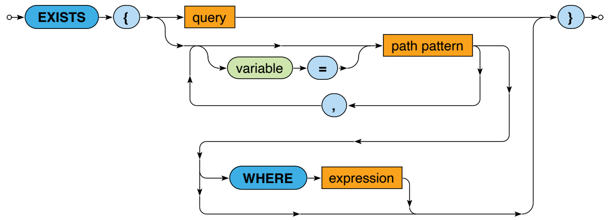


- Satisfied if and only if a **non-empty result** is yielded
- E.g.: `(m)-[:PLAY]->(:ACTOR)`
  - Ensures the existence of at least one actor for an already resolved movie node `m`

Search Conditions

Existential subquery conditions

- **Subquery** with **top-level query** expressive power
- Or standard **graph pattern** with optional filtering

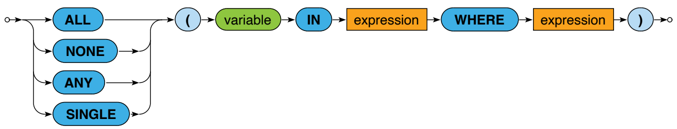


- Satisfied if and only if a **non-empty result** is yielded
- E.g.: `EXISTS { (m)-[:PLAY]->(a:ACTOR) WHERE a.name = "Ivan Trojan" }`

Search Conditions

Quantifier conditions

- Allow to simulate quantifiers and their derivatives



- Satisfied if and only if…
  - `ALL`: **all items** satisfy a given condition
  - `NONE`: **no item** satisfies a given condition
  - `ANY`: **at least one item** satisfies a given condition
  - `SINGLE`: **exactly one item** satisfies a given condition
- E.g.: `ANY (g IN m.genres WHERE g = "comedy")`

Search Conditions

Logical conditions

- Standard logical connectives are available
  - `AND` (**conjunction**)
  - `OR` (**disjunction**)
  - `NOT` (**negation**)

Match Clause

`OPTIONAL` **mode** of `MATCH` clauses

- **Optionally** attempts to find **matching data sub-graphs**...
  - When not possible, one solution with all variables bound to `null` is generated
- **Left outer natural join** is used when chaining

Match Clause: Example

Movies from *2005* or earlier, optionally their actors born after *1965*

```
MATCH (m:MOVIE)
  WHERE (m.year <= 2005)
OPTIONAL MATCH (m)-[:PLAY]->(a:ACTOR)
  WHERE (a.year > 1965)
RETURN m.title, a.name
```

m
(samotari)
(stesti)

⟕

m	i
(vratnelahve)	(machacek)
(samotari)	(machacek)
(samotari)	(schneiderova)
(medvidek)	(machacek)

=

m	a
(samotari)	(machacek)
(samotari)	(schneiderova)
(stesti)	`null`

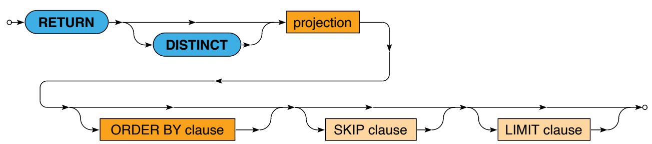
→

m.title	a.name
"Samotáři"	"Jiří Macháček"
"Samotáři"	"Jitka Schneiderová"
"Štěstí"	`null`

Return Clause

`RETURN` **clause**

- Defines the final **query result** to be **returned to the user**
  - Can only be provided as the **very last clause** in the chain



- `DISTINCT` **modifier**: removes **duplicate** solutions
- `ORDER BY` **subclause**
- `SKIP` and `LIMIT` **subclauses**: **pagination** of solutions

Return Clause: Example

Actors born in *1965* or later and numbers of movies they played in

```
MATCH (a:ACTOR)
  WHERE (a.year >= 1965)
RETURN a.name, SIZE([ (a)<-[:PLAY]-(m:MOVIE) | m ]) AS count
  ORDER BY count DESC
```

a
(schneiderova)
(machacek)
(vilhelmova)

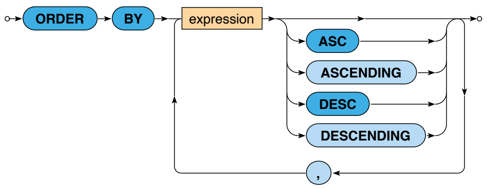
→

a.name	count
"Jiří Macháček"	3
"Jitka Schneiderová"	1
"Tatiana Vilhelmová"	0

Solution Modifiers

ORDER BY **subclause**

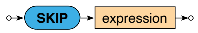
- Defines the **order of solutions** within the query result
  - Multiple criteria can be specified
    - Nodes, relationships, nor paths cannot be used for this purpose
  - The order is undefined unless explicitly defined
- **Default direction** is ASC



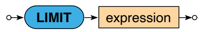
Solution Modifiers

Pagination

- SKIP **subclause**
  - Determines the **number of solutions to be skipped** in the query result



- LIMIT **subclause**
  - Determines the **number of solutions to be included** in the query result



Grouping and Aggregation: Example

Actors born in *1965* or later and movies they played in (cont'd)

```
MATCH (a:ACTOR)
  WHERE (a.year >= 1965)
OPTIONAL MATCH (a)<-[:PLAY]-(m:MOVIE)
RETURN a.name, COUNT(m) AS count, COLLECT(m.title) AS movies
```

a
(machacek)
(schneiderova)
(vilhelmova)

→

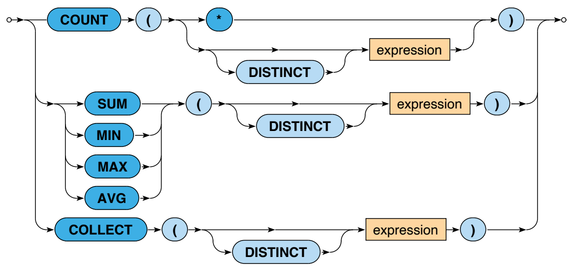
a	m
(machacek)	(vratnelahve)
(machacek)	(samotari)
(machacek)	(medvidek)
(schneiderova)	(samotari)
(vilhelmova)	null

→

a.name	count	movies
"Jiří Macháček"	3	["Vratné lahve", "Samotáři", "Medvídek"]
"Jitka Schneiderová"	1	["Samotáři"]
"Tatiana Vilhelmová"	0	[ ]

Grouping and Aggregation

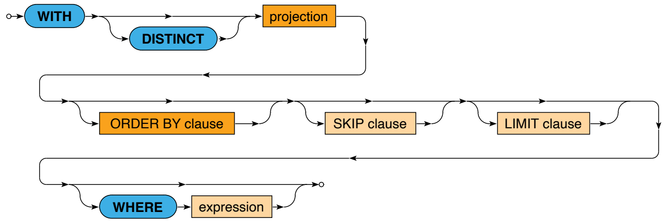
Aggregate functions



With Clause

`WITH` **clause**

- **Constructs** another **intermediate result** in the chain
  - Analogous behavior to the `RETURN` clause
    - Except that no output is sent to the user
  - Optional `WHERE` **subclause** can also be provided



With Clause: Example

Movies with above average number of actors and rating at least *75*

```
MATCH (m:MOVIE)
WITH m, SIZE([(m)-[:PLAY]->(a:ACTOR)|a]) AS actors
WITH AVG(actors) AS average
MATCH (m:MOVIE)
  WHERE (SIZE([(m)-[:PLAY]->(a:ACTOR)|a]) > average) AND (m.rating >= 75)
RETURN m.title, m.rating
```

m
(vratnelahve)
(samotari)
(medvidek)
(stesti)

→

m	actors
(vratnelahve)	2
(samotari)	3
(medvidek)	2
(stesti)	0

→

average
1.75

→

average	m
1.75	(vratnelahve)
1.75	(samotari)

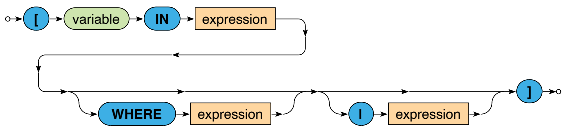
→

m.title	m.rating
"Vratné lahve"	76
"Samotáři"	84

List Operations

List comprehension

- **Creates a new list based on items of an existing list**
  - Only items satisfying a given condition are considered
  - New output items can be constructed
    - Otherwise the original ones are returned intact

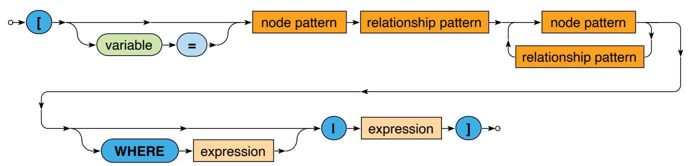


- Examples
  - `[ i IN range(1, 5) WHERE i % 2 = 0 ]` → `[2, 4]`
  - `[ i IN range(1, 5) WHERE i % 2 = 0 | i * 10 ]` → `[20, 40]`

List Operations

Pattern comprehension

- **Creates a new list based on solutions of a given path pattern**
  - Only solutions satisfying a given condition are considered



- Example
  - `[ (m:MOVIE)-[:PLAY]->(a:ACTOR) WHERE (m.year >= 2005) AND (a.name = "Jiří Macháček") | m.title ]` → `["Vratné lahve", "Medvídek"]`

Path Expressions: Examples

Absolute path expressions

```
/
```

```
/movies
```

```
/movies/movie
```

```
/movies/movie/title/text()
```

```
/movies/movie/@year
```

Relative path expressions

```
actor/text()
```

```
@director
```


Path Expressions: Axes

`child` axis

- Selects **children** of a given context node
  - Note that attributes are not considered to be child nodes!
- Used as the **default axis** (when omitted)

```
/movies/child::movie
```

`attribute` axis

- Selects **attributes** of a given context node
  - Note that this is the <u>only</u> axis that can select attributes!

```
/movies/movie/attribute::year
```

`self` axis

- Selects just the current **context node**

Path Expressions: Axes

`descendant`(`-or-self`) axes

- Select all (non-attribute) **nodes in a subtree** of a given context node excluding / including itself

```
/descendant::actor/text()
```

`parent` axis

- Selects the **parent node** of a given context node

`ancestor`(`-or-self`) axes

- Select all **ancestors** of a given context node
  - I.e., the parent, the parent of the parent, and so on, until the document node, excluding / including the context node itself

Path Expressions: Axes

`preceding-sibling` and `following-sibling` axes

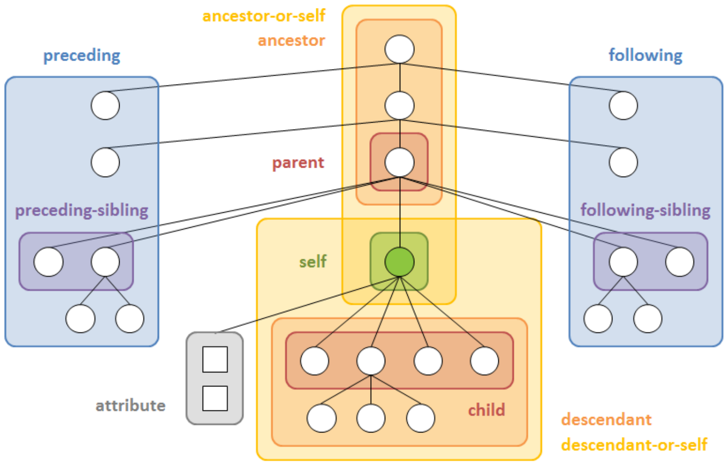
- Select all **siblings** of a given context node that occur before / after this context node in the document order

```
/descendant-or-self::movie/title/following-sibling::actor
```

`preceding` and `following` axes

- Select all (non-attribute) **nodes that occur before / after** a given context node in the document order, <u>excluding</u> nodes returned by the `ancestor` / `descendant` axis

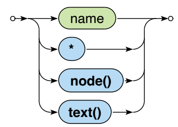
Path Expressions: Axes



Path Expressions: Node Tests

Node test

- Filters the nodes selected by the axis using a basic condition
  - Only **names and kinds** of nodes can be tested



`name`: elements / attributes with a given name

```
/movies
```

```
/movies/movie/attribute::year
```


Path Expressions: Node Tests

`*`: all elements / attributes

```
/movies/*
```

```
/movies/movie/attribute::*
```

`text()`: all text nodes

```
/movies/movie/title/text()
```

`node()`: all nodes

```
/movies/descendant-or-self::node()/actor
```


Path Expressions: Predicates

Path existence tests

- Relative or absolute path expressions
  - Relative path expressions are evaluated with respect to the node for which a given predicate is tested
- Treated as `true` when evaluated to a **non-empty sequence**

```
/movies/movie[actor]
```

```
/movies/movie[actor]/title/text()
```

Comparisons

- General, value, or node comparison expressions

```
/descendant::movie[@year > 2000]
```

```
/descendant::movie[count(actor) ge 3]/title
```

Path Expressions: Predicates

Position tests

- Allow for filtering of items based on context positions
  - Numbered starting with 1
  - Always relative to the current context (intermediate result)
  - Base order is implied by the axis used

```
/descendant::movie/actor[position() = 1]
```

```
/descendant::movie[actor][position() = last()]
```

Logical expressions

- `and`, `or`, `not` connectives

```
/movies/movie[@year > 2000 and @director]
```

```
/movies/movie[@director][@year > 2000]
```


Path Expressions: Abbreviations

Omitted axis: the default `child` axis is assumed

```
/movies/movie/title
```

```
/child::movies/child::movie/child::title
```

Attributes: `@` ⇔ `attribute::`

```
/movies/movie/@year
```

```
/movies/movie/attribute::year
```

Descendants: `//` ⇔ `/descendant-or-self::node()/`

```
/movies//child::actor
```

```
/movies/descendant-or-self::node()/child::actor
```


Path Expressions: Abbreviations

Context item: `.` ⇔ `self::node()`

```
/movies/movie[.//actor]
```

```
/movies/movie[self::node()//actor]
```

Parent: `..` ⇔ `parent::node()`

Position tests: `[number]` ⇔ `[position() = number]`

```
/movies/movie/child::actor[2]
```

```
/movies/movie/child::actor[position() = 2]
```

```
/movies/movie[actor][last()]
```

```
/movies/movie[actor][position() = last()]
```


Comparison Expressions

Comparisons

- **General** comparisons
  - Two sequences of values are expected to be compared
  - `=`, `!=`, `<`, `<=`, `>=`, `>`
  - E.g.: `(0,1) = (1,2)`
- **Value** comparisons
  - Two standalone values (singleton sequences) are compared
  - `eq`, `ne`, `lt`, `le`, `ge`, `gt`
  - E.g.: `1 lt 3`
- **Node** comparisons
  - `is` – tests identity of nodes
  - `<<`, `>>` – test positions of nodes (preceding, following)
  - Similar behavior as in the case of value comparisons

Expressions

XQuery expressions

- **Path** expressions (traditional XPath)
  - Selection of nodes of an XML tree
- **FLWOR** expressions
  - `for … let … where … order by … return …`
- **Conditional** expressions
  - `if … then … else …`
- **Quantified** expressions
  - `some|every … satisfies …`

Node Constructors

Constructors

- Allow for **creation of new nodes** for elements, attributes, …
  - I.e. nodes that do not exist in the original XML document

Direct constructor

- Well-formed XML fragment with embedded query expressions
  - E.g.: `<movies>{ count(//movie) }</movies>`

Computed constructor

- Special syntax
  - E.g.: `element movies { count(//movie) }`

Node Constructors: Example

Create a summary of all movies

```
<movies>
  <count>{ count(//movie) }</count>
  {
    for $m in //movie
    return
      <movie year="{ data($m/@year) }">{ $m/title/text() }</movie>
  }
</movies>
```

```
<movies>
  <count>3</count>
  <movie year="2006">Vratné lahve</movie>
  <movie year="2000">Samotáři</movie>
  <movie year="2007">Medvídek</movie>
</movies>
```


Node Constructors: Example

Create a summary of all movies

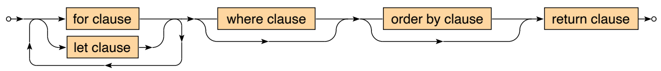
```
element movies {
  element count { count(//movie) },
  for $m in //movie
  return
    element movie {
      attribute year { data($m/@year) },
      text { $m/title/text() }
    }
}
```

```
<movies>
  <count>3</count>
  <movie year="2006">Vratné lahve</movie>
  <movie year="2000">Samotáři</movie>
  <movie year="2007">Medvídek</movie>
</movies>
```

FLWOR Expressions

FLWOR expression (XQuery 1.0)

- Allow for advanced **iterations over sequences** of items



Clauses

- `for` – selection of items to iterate over
- `let` – bindings of auxiliary variables
- `where` – conditions to be satisfied
- `order by` – order in which the items are processed
- `return` – result to be constructed

FLWOR Expressions: Example

Find titles of movies with rating *75* and more

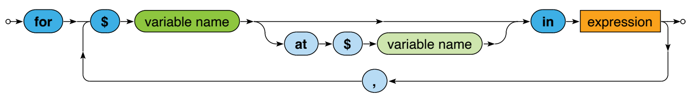
```
for $m in //movie
let $r := $m/@rating
where $r >= 75
order by $m/@year
return $m/title/text()
```

```
Samotáři
Vratné lahve
```


FLWOR Expressions: Clauses

For clause

- **Iterates over items** of one or more input sequences
  - These items are accessible via the introduced variables

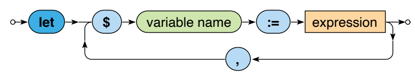


- Optional **positional variable**
  - Allows to access the ordinal number of the current item
- When **multiple input sequences** are provided…
  - Then the behavior is identical to the usage of multiple consecutive single-variable for clauses
    - I.e., as if the for loops are embedded into each other

FLWOR Expressions: Clauses

Let clause

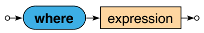
- Defines one or more auxiliary **variable assignments**



FLWOR Expressions: Clauses

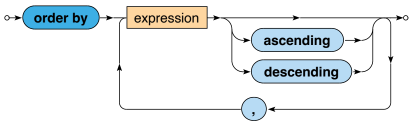
Where clause

- Allows to describe complex **filtering conditions**
- Items not satisfying the conditions are skipped



Order by clause

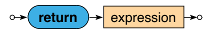
- Defines the **order in which the items are processed**



FLWOR Clauses

Return clause

- **Defines how the result sequence is constructed**
- Evaluated once for each suitable item



Various supported **use cases**

- Querying, joining, grouping, aggregation, integration, transformation, validation, …

FLWOR Examples

Find titles of movies filmed in *2000* or later such that they have at most 3 actors and a rating above the overall average

```
let $r := avg(//movie/@rating)
for $m in //movie[@rating >= $r]
let $a := count($m/actor)
where ($a <= 3) and ($m/@year >= 2000)
order by $a ascending, $m/title descending
return $m/title
```

```
<title>Vratné lahve</title>
<title>Samotáři</title>
```


FLWOR Examples

Find movies in which each individual actor stared

```
for $a in distinct-values(//actor)
return <actor name="{ $a }">
         {
           for $m in //movie[actor[text() = $a]]
           return <movie>{ $m/title/text() }</movie>
         }
       </actor>
```

```
<actor name="Zdeněk Svěrák">
  <movie>Vratné lahve</movie>
</actor>
<actor name="Jiří Macháček">
  <movie>Vratné lahve</movie>
  <movie>Samotáři</movie>
  <movie>Medvídek</movie>
</actor>
...
```

FLWOR Examples

Construct an HTML table with data about movies

```
<table>
  <tr><th>Title</th><th>Year</th><th>Actors</th></tr>
  {
    for $m in //movie
    return
      <tr>
        <td>{ $m/title/text() }</td>
        <td>{ data($m/@year) }</td>
        <td>{ count($m/actor) }</td>
      </tr>
  }
</table>
```

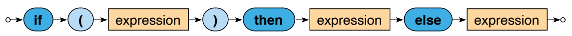

FLWOR Examples

Construct an HTML table with data about movies

```
<table>
  <tr><th>Title</th><th>Year</th><th>Actors</th></tr>
  <tr><td>Vratné lahve</td><td>2006</td><td>2</td></tr>
  <tr><td>Samotáři</td><td>2000</td><td>3</td></tr>
  <tr><td>Medvídek</td><td>2007</td><td>2</td></tr>
</table>
```


Conditional Expressions

Conditional expression



- Note that the `else` branch is compulsory
  - Empty sequence `()` can be returned if needed

Example

```
if (count(//movie) > 0)
then <movies>{ string-join(//movie/title, ", ") }</movies>
else ()
```

```
<movies>Vratné lahve, Samotáři, Medvídek</movies>
```


Quantified Expressions

Examples

Find titles of movies in which *Ivan Trojan* played

```
for $m in //movie
where
  some $a in $m/actor satisfies $a = "Ivan Trojan"
return $m/title/text()
```

```
Samotáři
Medvídek
```

Find names of actors who played in all movies

```
for $a in distinct-values(//actor)
where
  every $m in //movie satisfies $m/actor[text() = $a]
return $a
```

```
Jiří Macháček
```