CS698F Advanced Data Management

Instructor: Medha Atre

Recap

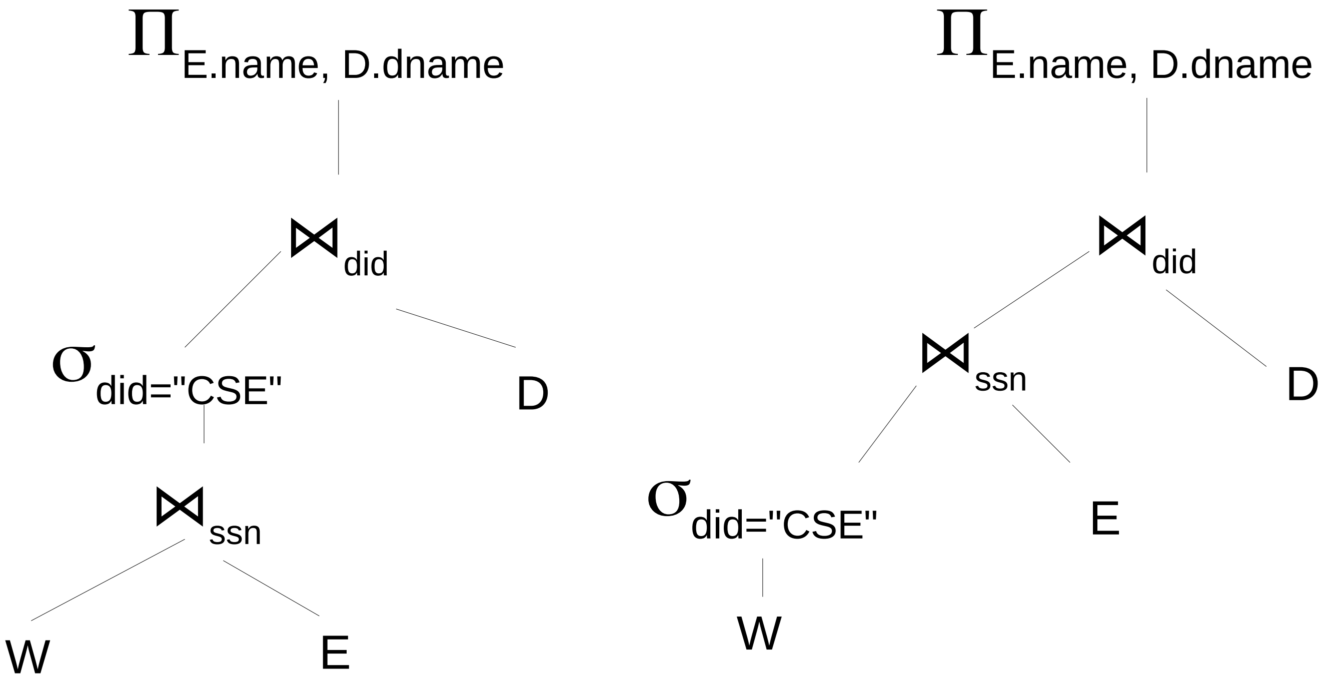
- Query rewriting a.k.a. considering various query plans for the *same effective results*.
  - Relational algebraic equivalences help
- Indexes on the tables a.k.a. access methods
  - Types of indexes – B+ trees, Hash index, others we will see in the contexts of different data types.
- Join methods and their costs
  - Nested-loop, sort-merge, index-nested-loop join, hash join etc.
- Finally combining the above two together for cost optimization.

Types of joins

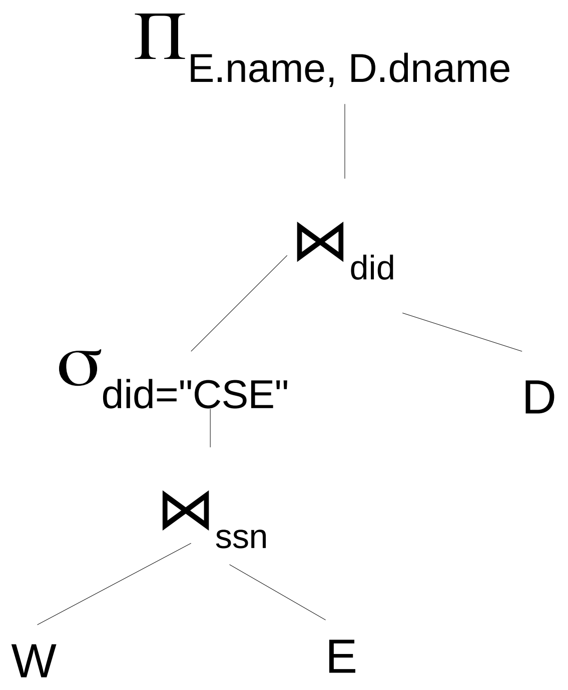
- Block-nested-loop join
  - When none of the tables have indexes and none of them are sorted on the join attributes.
- Index-nested-loop join
  - When one relation has an index on the join attribute.
- Merge-join
  - When both the relations have respective indexes on the joined attributes.
- Sort-merge-join
  - Sort both the relations on the join attribute first and then merge.
- Hash-join
  - Partition the attribute values from both the tables into *k* buckets and then join pairwise bucket.

Cost estimation in detail

SELECT E.name, D.dname
FROM WorksIn2 as W, Employees as E, Department as D
WHERE
W.did="CSE" **AND** W.did=D.did **AND** W.ssn=E.ssn

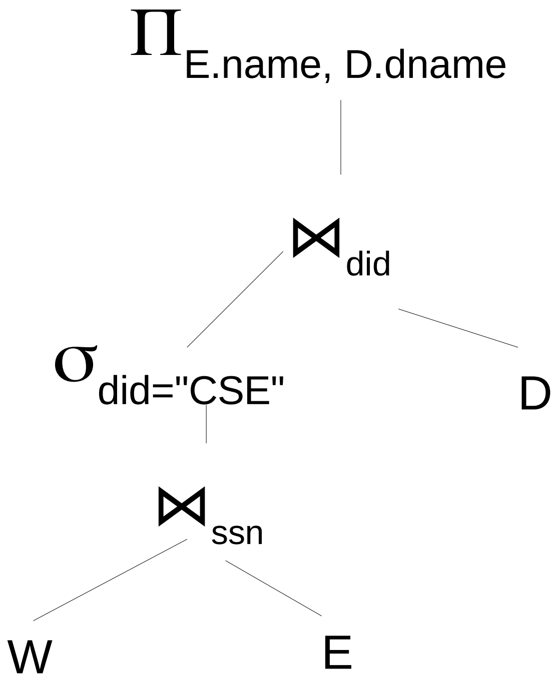


Cost estimation



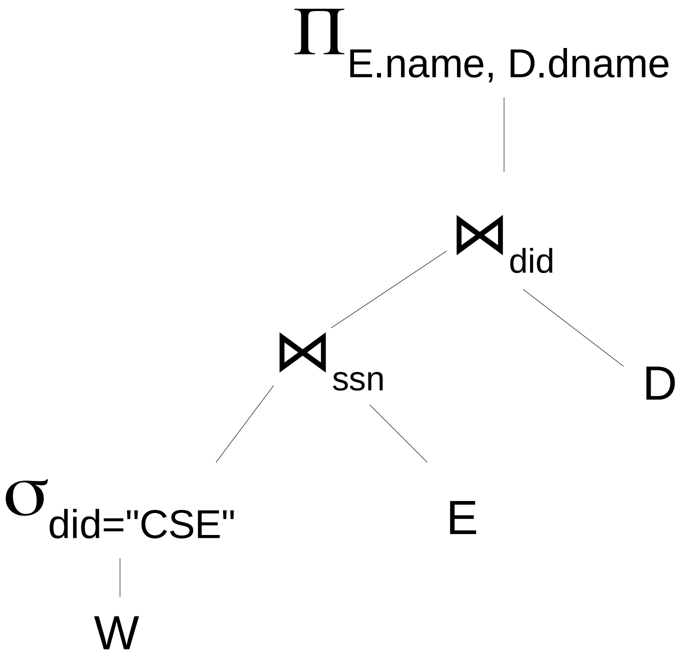
- Consider first join $\bowtie_{ssn}$
  - Does W and E have an index on *ssn*?
  - Is *ssn* the primary key of any of the relations?
- From the above, estimate the number of tuples to be processed.
- Using #tuples (in turn #pages), consider various join methods
  - Estimate the cost of various joins
  - Pick the least cost one store this cost in a dynamic prog memoization table!

Cost estimation



- Move a step higher – selection condition $\sigma_{did="CSE"}$
  - No index, tuplewise scan over temp table of prev join
  - Cost: No added cost!
    - Why?
- #tuple estimates: thumb rule 1/10 * #tuple estimates from prev join.
- Move a step higher – join condition
  - Similar analysis as previous join

Cost estimation

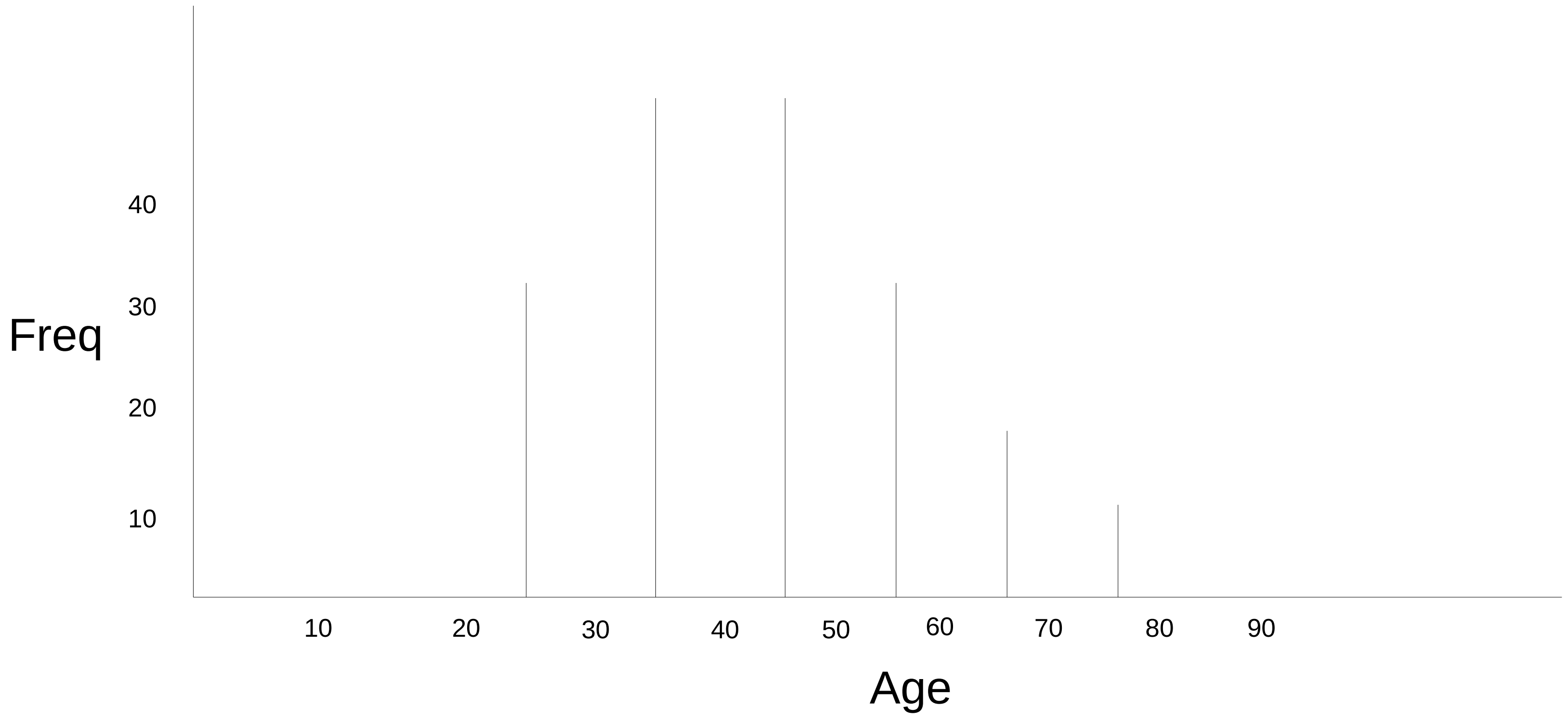


- Consider $\sigma_{did="CSE"}$
  - Does *W* have an index on *did*?
- #tuple estimates:
  - If index: exact # tuples
  - If not: 1/10 of all tuples
- Move a step higher – join $\bowtie_{ssn}$
  - Take #tuples estimate from prev selection
  - Does E have an index on ssn?
  - Consider various join plans with #tuples from W *after* selection and from E (depending on if index or not)
- So on…..

Thumbrules!

- Thumbrules *WHERE* clause:
- Column = value (selection)
  - If index: 1/Nkeys(I) * #tuples, no index: 1/10 * #tuples
  - Assumes *uniform* distribution of unique values.
  - E.g., if a table as 1000 rows and a column "B" has 10 unique values, 1000/10 = 100 rows contain the same value in column "B"
- What if distribution is *non-uniform*?
  - Either do estimation with the uniform distribution assumption – error prone
  - Maintain *histograms*, and estimate by searching in the histogram ranges.

Histograms



Thumbrules!

- Column1 = column2 (join)
  - If index on both: *1/max(Nkeys(I1), Nkeys(I2)) * #tuples(T1) * #tuples(T2)*
  - Let tuples in *T1* be *M* and in *T2* be *N* and unique values of join column "*I*" in T1 be *Nkeys(I1)*, and in T2 be *Nkeys(I2)*
  - Each unique value repeats M ÷Nkeys(I1) times in T1
  - Each unique value repeats N ÷Nkeys(I2) times in T2

Thumbrules!

- Column1 = column2 (join)
  - Join is essentially *{Nkeys(I1)} ∩ {Nkeys(I2)}.* Assume *{Nkeys(I1)} ⊆ {Nkeys(I2)}* => worst case maximum join results!
  - Each unique value in I1, generates:
    (M ÷ Nkeys(I1)) x (N ÷ Nkeys(I2)) results.
  - There are Nkeys(I1) such unique values.
  - Hence total results = $\text{Nkeys(I1)} \times \frac{N}{\text{Nkeys(I2)}} \times \frac{M}{\text{Nkeys(I1)}}$

Thumbrules!

- Column1 = column2 (join)
  - Hence total results = $\dfrac{N}{Nkeys(I2)} \times \dfrac{M}{}$

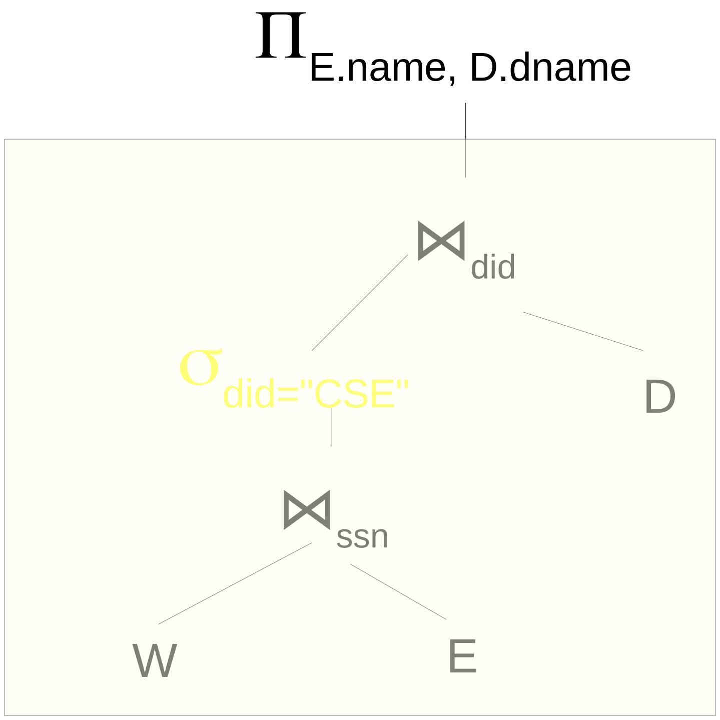
$$= \frac{N \times M}{Nkeys(I2)}$$

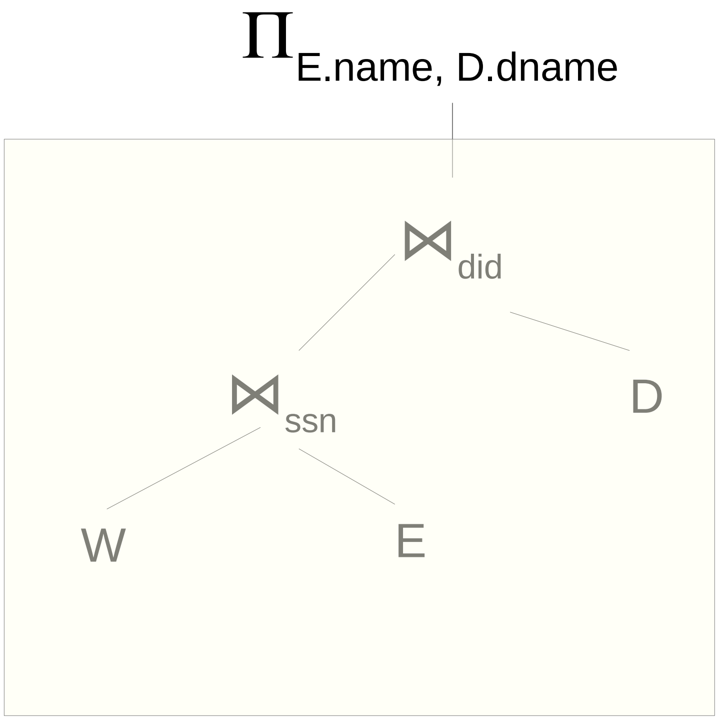
$$= \frac{N \times M}{max(Nkeys(I2), Nkeys(I1))}$$

Other Improvements

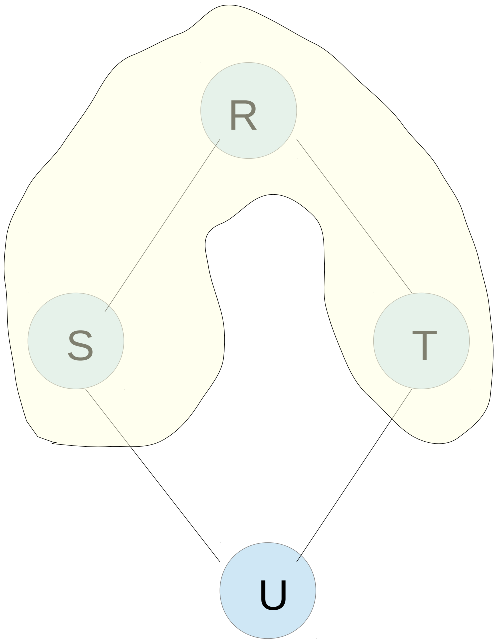
- Join tables:
  - If some joins are observed to be frequent, preserve their join results.
  - Mining into the query logs, and pattern recognition!

Pattern recognition in queries

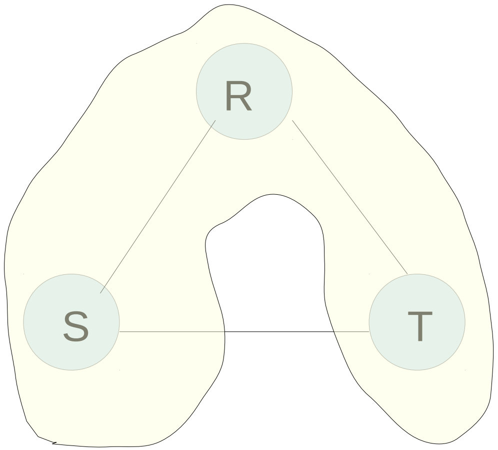




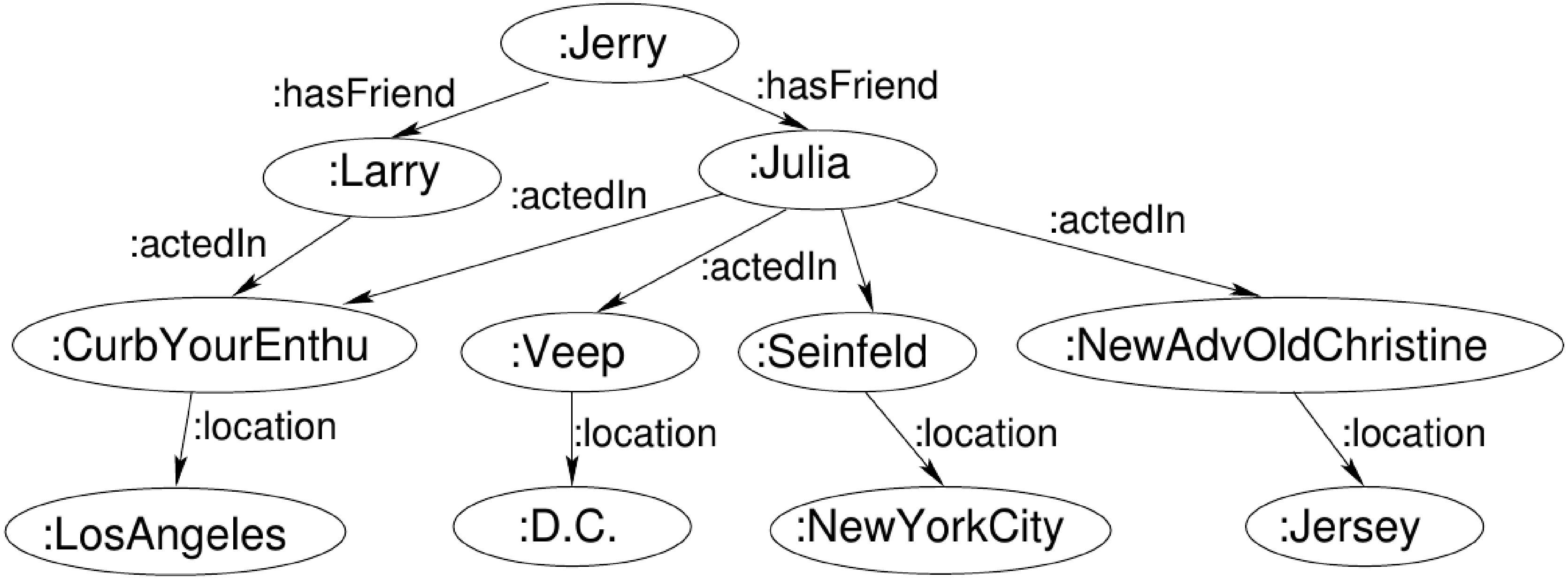
Pattern recognition in queries



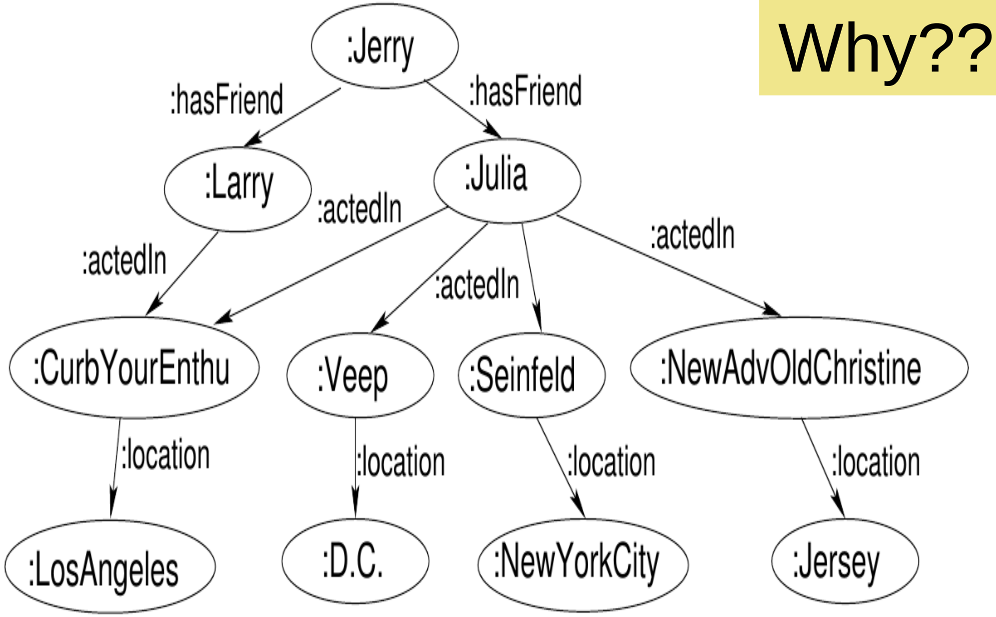
- Techniques like approximate pattern match.
- Subgraph isomorphism
- Since query graphs are very small NP properties do not matter.
- One time activity!



Graphs!


:Jerry
:hasFriend
:hasFriend
:Larry
:Julia
:actedIn
:actedIn
:actedIn
:actedIn
:CurbYourEnthu
:Veep
:Seinfeld
:NewAdvOldChristine
:location
:location
:location
:location
:LosAngeles
:D.C.
:NewYorkCity
:Jersey

Graph as a table

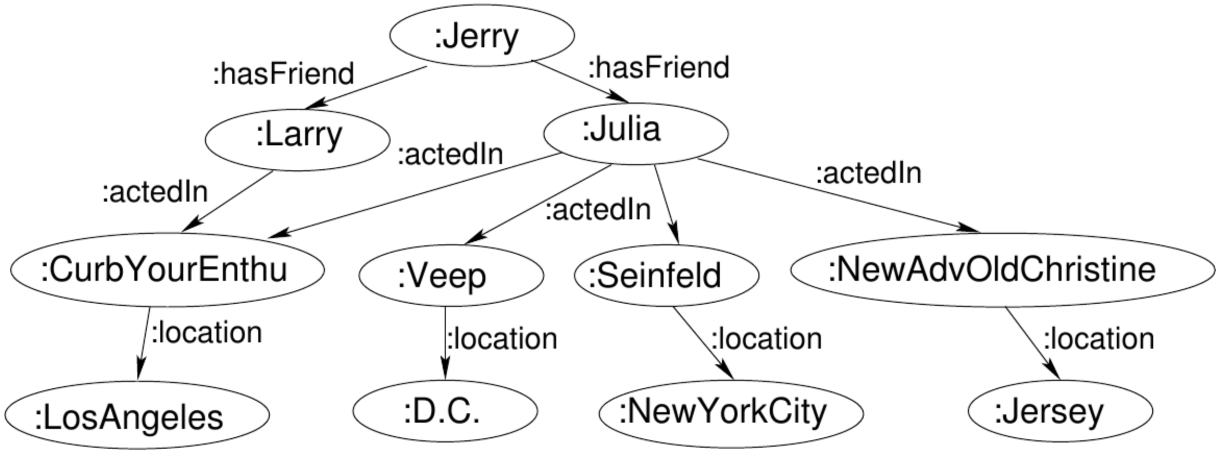


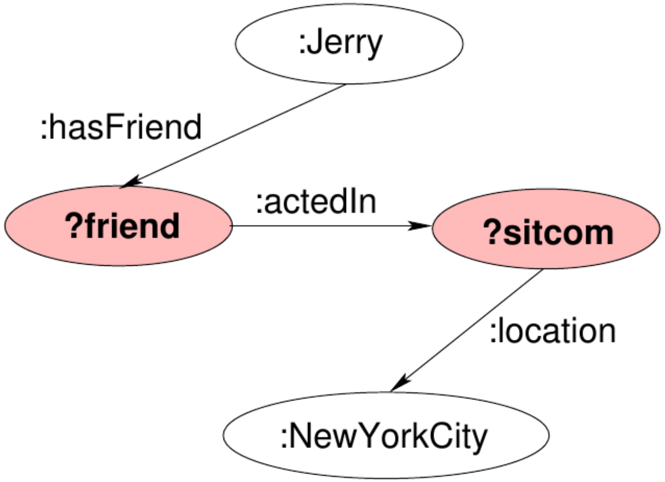
Why??

RDF table

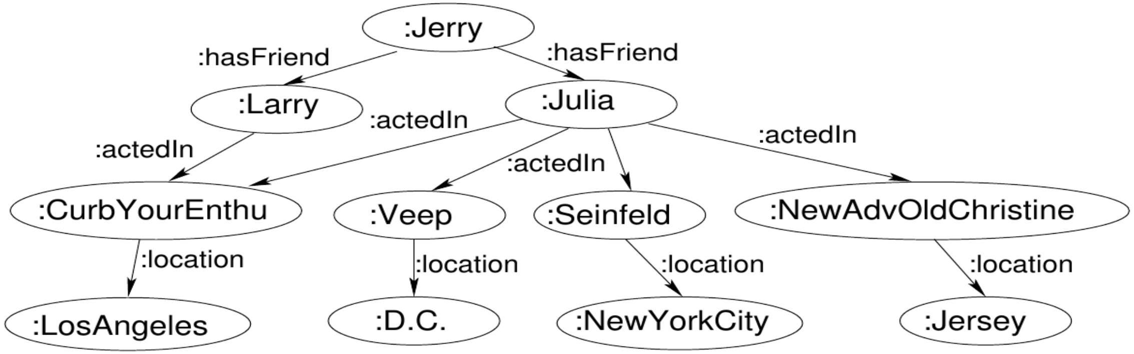
S	P	O
:Jerry	:hasFriend	:Larry
:Jerry	:hasFriend	:Julia
:Larry	:actedIn	:CurbYourEnthu
:Julia	:actedIn	:Seinfeld
:Julia	:actedIn	:Veep
:Julia	:actedIn	:NewAdvOldChristine
:Julia	:actedIn	:CurbYourEnthu
:Seinfeld	:location	:NewYorkCity
:Veep	:location	:D.C.
:CurbYourEnthu	:location	:LosAngeles
:NewAdvOldChristin	:location	:Jersey

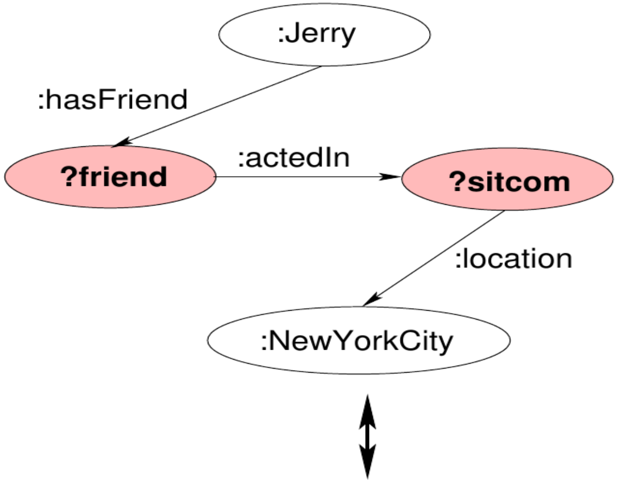
A graph pattern query





A graph pattern query





SPARQL BGP

```
SELECT ?friend ?sitcom
WHERE {
      :Jerry :hasFriend  ?friend .
      ?friend :actedIn ?sitcom .
      ?sitcom :location :NewYorkCity .
}
```

Pattern query as a self-join

SQL inner–join

```
SELECT t1.o, t2.o
FROM RDF as t1, RDF as t1,
RDF as t3
WHERE
t1.S=":Jerry" AND
t1.P=":hasFriend" AND
t1.O=t2.S AND t2.O=t3.O
AND t2.P=":actedIn" AND
t3.P=":location" AND
t3.O=":NewYorkCity"
```

↔

SPARQL BGP

```
SELECT ?friend ?sitcom
WHERE {
    :Jerry :hasFriend  ?friend .
    ?friend :actedIn ?sitcom .
    ?sitcom :location :NewYorkCity .
}
```

Processing pattern queries

- Treat a graph database as a single giant table
  - Three columns, S, P, O, s = subject (source node), P = property (edge-label), O = object (destination node)
- A pattern query makes several self-joins on the table
  - *Selection conditions* are the fixed values on edges or nodes of the pattern
  - Treat each edge with its selection conditions as a separate table.
- However, I/O costs will be *lower* than the actual relational tables.
  - Why?

Indexing methods

- Every graph table is a fixed 3-column table
  - How many maximum indexes are possible on it?
- Why create all possible indexes?
  - Would they be clustered or unclustered?
- What to do if we want multiple clustered indexes?
  - Data duplication?
- Does this table have a schema?
- Does it have a primary key, foreign key?