Advanced Data Management

Medha Atre

Office: KD-219

atrem@cse.iitk.ac.in

Aug 1, 2016

Recap

- Overview of the relational and graph shaped data.
- Overview of the general purpose tools like Hadoop, SPARK.
- Special purpose tools for the relational data, like MySQL, PostgreSQL, Many other commercial stores like IBM DB2, MonetDB, Virtuoso, Cassandra, Cloudera, BigTable etc.
- Special purpose graph processing tools like BitMat, RDF-3X, Neo4j, HypergraphDB, Pregel, Trinity etc.

Query Optimization

- Involves choosing a query evaluation plan that reduces the total cost of the query execution.
- Cost includes:
  - Number of disk blocks/tuples to access.
  - Number of times a relation/block has to be read.
  - Any intermediate query results, their size, and cost of writing them to the disk (disk spooling) if required.
- The simplest way – create indexes.
- Push *selection predicates* as down as possible.

  SELECT PRODUCT.Description, PRODUCT.Brand WHERE

  <u>STORE.City="New York"</u> AND

  STORE.Store_key=SALES_FACT.Store_key AND

  SALES_FACT.Product_key=PRODUCT.Product_key
- Push *projections* as down as possible.

Query Optimization

- Consider various permutations of joins and other operators *without* affecting the correctness of the results – use commutativity, associativity, and distributive properties of joins and other operators such as selections and projections.
- Histograms over unique column values.
- Various permutations of joins – similar to how many binary trees with n leaves, where n is the number of tables to be joined – Catalan number $C_{n-1} = \frac{(2n-2)!}{n!(n-1)!}$
- E.g., for a join of 7 tables (or 7 self-joins as common in graph data), number of join trees to be considered are 132!

IBM's System R optimizer

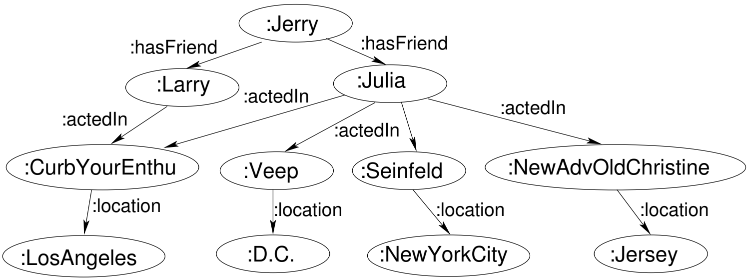
- Use database *statistics*, e.g., number of tuples in a table, columnwise cardinality, distribution of unique values (a.k.a. histograms), available indexes, index size, index range etc.
- Consider only *left* or *right deep* join trees.
- Focus on class of SQL queries without nesting.
- Not to perform duplicate elimination while projecting out the results, unless the query has a DISTINCT clause.
- Account of CPU as well as I/O cost.

Estimating the result size of a join query is an NP-complete problem! For more information see [NgoPODS2012, AtseriasFOCS2008].

Graphs and Relational Algebra

Since graphs can be stored as a relational table, and graph pattern queries can be translated as SQL join queries, all the relational algebra hardness results and query optimization techniques generally apply to graph pattern queries too.

Recap


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

Graph data and queries

Data

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

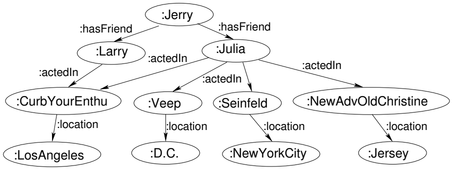
SPARQL

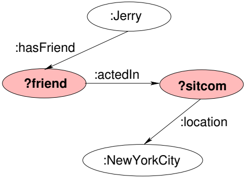
SELECT ?friend ?sitcom **WHERE** {
:Jerry :hasFriend **?friend** .
?friend :actedIn **?sitcom** .
?sitcom :location :NewYorkCity .
}

Eqv. SQL query

SELECT t1.o, t2.o from rdf as t1, rdf as t2, rdf as t3 **WHERE** t1.s=":Jerry" and t1.p=":hasFriend" and t2.p=":actedIn" and t3.p=":location" and t3.o=":NewYorkCity" and **t1.o=t2.s and t2.o=t3.s**

Graphical Representation





Graphs – key problems

- Increasing size of graphs – from a few hundred million to over a billion triples, e.g., DBPedia has $\approx$ 600 million triples, Linked Open Data project over 30 billion triples.
- While the size of the secondary memory (hard disk) has increased from a few hundred GBs to a few TBs, the size of the typical main memory still remains at a few GBs.
- Low selectivity queries – those which access a large amount of data and cannot benefit from fast *merge-joins*, e.g., queries with multiple joins on various attributes (join variables in case of SPARQL queries).
- Contemporary systems that do *pairwise pipelined (vectorized)* joins suffer in case of low-selectivity queries, due to skewed *cardinality* of attribute values.

Example of a low selectivity query

N	M
1	a
2	a
3	a
1	b
2	b
3	b
4	b
1	c
2	c
3	c
1	d
2	d
3	d

T1

⋈ T1.M=T2.M

13 tuples in T1

15 tuples in T2

28 tuples in all

M	S
a	1
a	2
a	3
a	4
b	1
b	2
b	3
c	1
c	2
c	3
c	4
c	5
c	6
c	7
c	8

T2

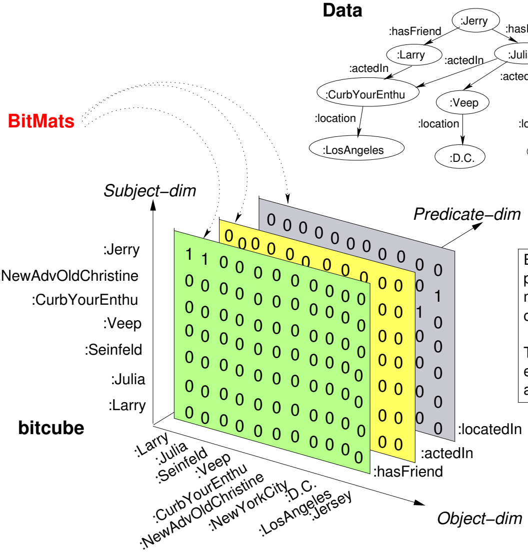
If we do a standard join of these two tables, we get **48** results (tuples) – a polynomial increase in the size of the results. This effect exacerbates for queries with multiple joins on different attributes, as is common with the RDF and SPARQL queries.

Instead, we want to find a way to keep the memory footprint of the query processor as low as possible to increase its scalability.

BitMat – key ideas

- A data structure based on compressed bit-vectors to represent RDF data called BitMat.
- Pattern matching algorithm that works directly on the compressed structure without uncompressing it.
- New way of representing a pattern query abstractly, especially for *wildcards* and *optional* pattern matches – Graph of Supernodes.
- Pre-join (pattern matching) pruning using the technique of *semi-joins*, thereby reducing the I/O overhead, and keeping a large amount of data in memory.
- Evaluation results on the popular RDF datasets of sizes up to 1.33 billions on a commodity laptop of 8 GB memory.

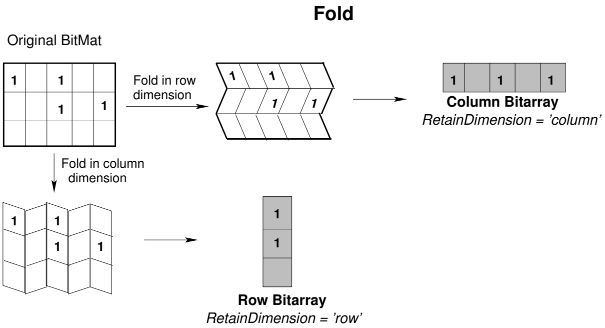
BitMat – brief overview



Each unique value of subjects, predicates, and objects in the data is mapped to the respective dimension of the bitcube.

This bitcube is then sliced along each dimension and the 2D BitMats are stored as the index structure.

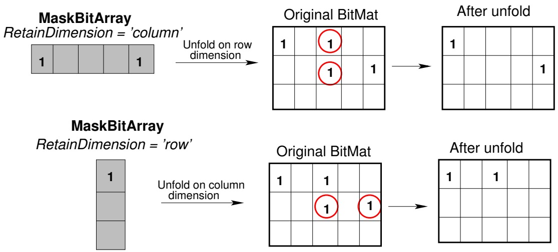
Fold and Unfold



fold(BM_{tp}, *RetainDimension*) procedure is nothing but projection of distinct values from the given dimension of BitMat, e.g., in the triple pattern (?friend :actedIn ?sitcom) if BM_{tp} is an O-S BitMat, then *?sitcom* is in the "row" dimension of the BitMat.

$$fold(BM_{tp}, dim_{?j}) \equiv \pi_{?j}(BM_{tp})$$

Unfold



For every *unset* bit in the *MaskBitArray*, *unfold(BM$_{tp}$, MaskBitArray, RetainDimension)* clears all the bits corresponding to that position of the *RetainDimension*.

$$unfold(BM_{tp}, \beta_{?j}, dim_{?j}) \equiv \{t \mid t \in BM_{tp}, t.?j \in \beta_{?j}\}$$

t is a triple in BM_{tp} that matches tp. $\beta_{?j}$ is the *MaskBitArray* containing bindings of $?j$ to be retained. $dim_{?j}$ is the dimension of BM_{tp} that represents $?j$, and $t.?j$ is a binding of $?j$ in triple t. In short, `unfold` keeps only those triples whose respective bindings of $?j$ are set to 1 in $\beta_{?j}$, and removes all other.

First Assignment

First assignment will be posted on the course webpage by the end of the day today. Due date will be August 15, 2016 midnight.

Please check the course webpage regularly for any important announcements, and assignment submission instructions.

www.cse.iitk.ac.in/users/atrem/courses/cs698f2016fall/